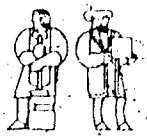
LABORATORY FOR COMPUTER SCIENCE



MASSACHUSETTS INSTITUTE OF TECHNOLOGY

MIT/LCS/TR-471

IMPLEMENTATION OF AN I-STRUCTURE MEMORY CONTROLLER

Kenneth M. Steele

March 1990

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TR-471	5. MONITORING ORGANIZATION REPORT NUMBER(S) N00014-84-K-0099
6a. NAME OF PERFORMING ORGANIZATION MIT Laboratory for Computer Science 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research/Department of Navy
6c. ADDRESS (City, State, and ZIP Code) 545 Technology Square Cambridge, MA 02139	7b. ADDRESS (City, State, and ZIP Code) Information Systems Program Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA/DOD 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Boulevard Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) Implementation of an I-Structure Memory Controller	
12. PERSONAL AUTHOR(S) Kenneth M. Steele	
13a. TYPE OF REPORT Technical 13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Year, Month, Day) March, 1990 15. PAGE COUNT 140
16. SUPPLEMENTARY NOTATION	

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	I-Structures, Dataflow, Memory Synchronization, Non-Busy-Waiting Locks, Monsoon Processor.

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Memory systems in large-scale parallel processors are characterized by high bandwidth requirements, and long access latency. Because many processors are issuing concurrent memory requests, requests can arrive at the memory in any order. Dataflow processors present an elegant and clean solution for synchronizing parallel tasks and tolerating long memory latency by providing hardware synchronization for individual instructions, single word task-continuations, and split-phase memory transactions. These same ideas are applied to the development of a high-bandwidth memory system with hardware support for synchronization. Synchronization does not cause busy-waiting or retrying; synchronization delays appear to the processors simply as long latency memory accesses.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Publications Coordinator	22b. TELEPHONE (Include Area Code) (617) 253-5894 22c. OFFICE SYMBOL

19. ABSTRACT (cont.)

I-structures are a method of memory access developed as an extension to the functional language Id to provide efficient data structures. Functional languages have the very important property of determinacy; program results are independent of the execution order of parallel tasks. By restricting each location to be write-once, data structures can remain determinate, regardless of the order of requests. The hardware stores, as lists, read requests of unwritten locations. These deferred requests are serviced automatically when the value is written. Different methods for storing lists of deferred requests are investigated.

Furthermore, this hardware synchronization can be used for Non-Busy-Waiting Locks, lazy data structures, and Imperative (non-synchronizing) operations. The implementation presented is specific to the Monsoon dataflow system; however, the design is applicable to other parallel processing systems, which require fast synchronization.

Implementation of an I-Structure Memory Controller

Kenneth M. Steele

MIT / LCS / TR-471
March 1990

This report describes research done at the Laboratory of Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-84-K-0099. This report was originally published as the author's master's thesis.

Implementation of an I-Structure Memory Controller

Kenneth M. Steele

Technical Report MIT / LCS / TR-471
March 1990

MIT Laboratory for Computer Science
545 Technology Square
Cambridge MA 02139

Abstract

Memory systems in large-scale parallel processors are characterized by high bandwidth requirements, and long access latency. Because many processors are issuing concurrent memory requests, requests can arrive at the memory in any order. Dataflow processors present an elegant and clean solution for synchronizing parallel tasks and tolerating long memory latency by providing hardware synchronization for individual instructions, single word task-continuations, and split-phase memory transactions. These same ideas are applied to the development of a high-bandwidth memory system with hardware support for synchronization. Synchronization does not cause busy-waiting or retrying; synchronization delays appear to the processors simply as long latency memory accesses.

I-structures are a method of memory access developed as an extension to the functional language Id to provide efficient data structures. Functional languages have the very important property of determinacy; program results are independent of the execution order of parallel tasks. By restricting each location to be write-once, data structures can remain determinate, regardless of the order of requests. The hardware stores, as lists, read requests of unwritten locations. These deferred requests are serviced automatically when the value is written. Different methods for storing lists of deferred requests are investigated.

Furthermore, this hardware synchronization can be used for Non-Busy-Waiting Locks, lazy data structures, and Imperative (non-synchronizing) operations. The implementation presented is specific to the Monsoon dataflow system; however, the design is applicable to other parallel processing systems, which require fast synchronization.

Key Words and Phrases: I-Structures, Dataflow, Memory Synchronization, Non-Busy-Waiting Locks, Monsoon Processor.

Acknowledgments

I would like to thank all of the people who have contributed to this report and my enjoyment of my time at MIT. I credit professor Arvind and his dataflow course (6.847) with sparking my interest in dataflow.

I also wish to thank the members, past and present, of the Computation Structures Group at MIT's Laboratory of Computer Science for making it an enjoyable, enlightening, and challenging place to work and a hard place to leave. I would particularly like to mention Richard Soley who was always willing, and able, to answer any question his officemate might throw at him and co-inventor of the Non-Busy-Waiting Locks presented in this report; Greg Papadopoulos, without whose Monsoon processor design, there would be no reason for this report; Bob Iannucci, who correctly convinced me that this was the right research group to join; Paul Barth, another co-inventor of Locks; Andy Boughton, without whom the group would not function and who kept the computer on which this report was written alive; Dana Henry, Madhue Sharma, Jamey Hicks, Chris Joerg, Janice Onanian, Jonathan Young, Ralph Tiberio and Jack Costanza for their help, support and friendship.

The people with whom I worked at Motorola in Tempe, Arizona over the summer played a key role in in making this a practical design, particularly Scott Cowling, Ralph Olson, Tom Bousman, and Jim Richey. There I truly learned the meaning of *hot* weather (120°F).

I am greatful to Cheri Stead, Bryan Butler and Doug Steele, all of whom contributed time and effort to make this report readable, and taught me some grammar along the way.

To my parents,
Douglas and Leslie Steele
whose loving support and
encouragement made this possible

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Multiprocessors

In the quest for ever faster computers, the idea of using multiple processors to solve a single problem is appealing. People do this all of the time when a team of people work together on a problem. As with people, multiprocessors require more communication than a single processor. To support the increased communication requirement, busses are replaced by networks. Networks have high bandwidth at the cost of increased complexity and longer latency. As the number of processors increases, the size and latency of the network increases. Networks also complicate the task of using caches to reduce latency, as is done in bus based systems. The problem is keeping the cached data consistent in a distributed system. All of these factors contribute to the conclusion that latency is an important issue in large multiprocessors.

What is hard about programming a multiprocessor? In a uniprocessor computer the order in which instructions are executed by the processor determines the order that operations are performed on the memory. This complete ordering determines the state of the memory. Time does not enter into the picture; the instructions can be run at any speed without affecting the final outcome. In a multiprocessor system the order in which instructions are executed by the processors no longer completely determines the order of operations performed on the memory. The relative timing of the processors, task scheduling, and network arbitration can all cause changes in the order in which operations arrive at the memory. This reordering can cause erroneous results unless special care is taken. The traditional solution is to provide programmers with a handful of synchronization primitives and let

them solve the problem. This adds to the complexity of programming multiprocessors and decreases their reliability by introducing more opportunities for error. The goal of the effort described in this report is to make programming multiprocessors easier by providing the programmer with a simpler memory model in which the order of operations does not matter. The final state of memory is determined only by the operations performed and not their order. I-structures are one method of order-independent synchronization. Hardware support is provided by the memory system to allow efficient synchronization.

The effects of synchronization are presented to the processors as simply a longer memory latency. This allows multiprocessors, which must already tolerate latency, to efficiently use I-structure synchronization methods. Four properties which help processors tolerate latency are: split-phase transactions, fast context switching, multiple outstanding requests to memory, and the ability to handle returned data out of order. Split-phase transactions separate the request to perform a memory operation from the return of data resulting from the operation. Thus, the request can be generated by a processor in a finite amount of time, independent of how long it takes for data to be returned. Fast context switching allows the processor to perform other useful work while waiting for the requested data. The context switching time must be shorter than the latency, or more time is wasted by switching to a new task. Multiple outstanding requests for data from the memory allows the processing of the requests to be pipelined, increasing potential throughput. Multiple requests also require the processors to accept returned data in any order. Returned data can get out of order because the latency of each request can be different. Dataflow processors have all four of these properties.

The problem of exploiting parallelism in programs and allocating the work to processors has been explored by many others and will not be dealt with here. Many years of research have gone into developing new languages to make programming parallel computers simpler. One such language, ID [12], is the basis for dataflow language research at MIT. I-structures are an integral part of ID.

The synchronization methods presented in this report have their roots in work performed with ID. Although this work was done in the context of dataflow computers, the ideas are not limited to dataflow computers.

1.2 Synchronizing Memory Design

The memory design presented in this report takes the synchronization primitives developed for the Monsoon [13] processor and applies them to provide hardware support for synchronization in a memory system. The hardware synchronization, although developed to support I-structures, is general enough to support other types of synchronization. Chapter 2 presents Non-Busy-Waiting Locks and support for lazy data structures. Non-synchronizing imperative operations (read and write), employed in conventional memory systems, are also supported. The design is to be built as the memory system for the Monsoon computer system. The practical requirements of implementation had a strong simplifying influence on the design.

1.3 I-Structure Black Box

Functionally, computer memory can be viewed as a black box into which requests are sent and from which values are returned. In a conventional memory, the read and write requests are processed in the order they arrive at the memory. I-structure memory differs in that it internally defers processing read requests for a location whose value has not been defined by a write. Locations are undefined until they are written. Once written, they cannot be rewritten. How the reads are deferred is not important to proper operation; it is, however, the most critical and difficult part of a practical design.

The I-structure controller must detect attempts to read locations before they are written. Again, how this detection is performed is an implementation issue, except that it does impose the functional requirement that each location is initialized (to undefined state) before it is used and then written, at most, once. Being write-once is the biggest limitation of I-structures.

1.4 Road Map

This report describes a hardware implementation of a synchronizing memory. Chapter 2 starts with a discussion of the issues in functional languages which I-structures were developed to solve. This discussion includes maintaining determinacy while avoiding copying,

providing a clear programmer's view of memory synchronization, and the effects on compilation. The remainder of the chapter presents the I-structure and Non-Busy-Waiting Locking mechanisms, with extensions for delayed triggers and imperative operations. It describes the two basic primitives needed to support these synchronization methods: presence bits and deferred lists. Chapter 3 covers three ways of storing deferred lists; two are local to the I-structure controller, and one is a distributed method involving the processors. The advantages of each list method are examined in terms of latency, network traffic, design complexity, and processor overhead. Appendix A contains the hardware specification used by Motorola to build the I-structure controller board for the Monsoon computer system. Implementation of the synchronization methods on that controller board is discussed in chapter 4, along with reasons for the design decisions.

1.5 Differences from Previous I-Structure Designs

I-structures are not a new idea; they have been discussed in many papers [2]. I-structures have been used in dataflow simulators for the ID language at MIT for many years. Steve Heller designed an I-structure memory controller (ISMC) [5] for the MIT Tagged Token Dataflow Architecture (TTDA) [1] for his master's thesis. That controller included hardware support for type checking and variable size memory objects. The controller was designed in a high level hardware description language but was not implemented; no TTDA processor was ever built.

The Sigma-1 [7] dataflow processor, built by the Electrotechnical Laboratory in Japan, includes a Structure Element (SE) that supports B-structures, which are identical to I-structures. Deferred lists are stored locally on each structure element using the local cons cell (two consecutive memory words) method presented in section 3.5. A Sigma-1 prototype was built with 128 processing element and 128 structure elements.

The design in this report is specifically intended for implementation. It is being built, along with the Monsoon processor [13], as a joint effort between MIT and Motorola Microcomputer Division in Tempe, Arizona. The design addresses most of the future work ideas for making an implementable controller suggested in Steve Heller's thesis. Hardware functionality is minimized to reduce cost and design time.

Chapter 2

Memory Synchronization Methods

Many different ways of synchronizing access to memory are possible. This chapter presents four of the methods that we have found useful in our work on the language ID for dataflow computers. These methods are: I-structures, Non-Busy-Waiting Locks, Delayed-Triggers and Imperative Operations. The usefulness of these memory synchronization methods is not limited to dataflow computers.

2.1 Language Background

The language ID is a "functional" language (with extensions) developed for programming parallel processors; parallelism is implicit in the definition of the language. Any statement can be run in parallel with any other statement or statements unless there is a data dependency between them. The data dependencies specify a partial ordering on the execution statements in the program. The computer may execute the statements in any order that satisfies this partial ordering. The property of determinacy requires that the same results be generated for any of the possible execution orders (ID is a determinate language). Determinacy makes programming parallel processors easier because the results are independent of how the program is executed. In a purely functional language, every statement and function must be without side-effects; it can not modify any existing data, only create new data. To update one value of an array, a function must create a new copy of the array, replace the old value with the new and return the new copy. This can be written as the function: (*update A i v*), which returns a new copy of the array A with the i^{th} element replaced with

v. Each element updated requires a new copy of the array. Consider the following piece of ID code which generates an array of Fibonacci numbers:

```
def fib_array n = { x0 = array (1,n);
                    x1 = (update x0 1 1);
                    x = (update x1 2 1);
                   in {for i ← 3 to n do
                       next x = (update x i (x[i-1] + x[i-2]));
                      finally x}};
```

Some explanation of ID syntax [12] is required. The symbol ← in the loop assigns the variable on the lefthand side to each value from the list on the righthand side in successive iterations. The statement "next x" defines the value of the variable x in the next iteration of the loop. The value returned from the loop is the value of x in the last iteration. The function makes n copies of the array, only the last of which is returned. This copying results in a large amount of overhead for running this simple program and it will only get worse for larger data structures. Also, the flow of the program has been sequentialized because the updated array must be passed to each statement in order.

Let us look at the *update* function in a different way. Instead of making a new copy of the array each time, we will keep an ordered list of the updates as index and value pairs. New updates are added to the beginning of the list, resulting in a a chronological ordering. If the array starts out undefined, then the list of updates completely specifies the array. To look up the value at an index, the list is searched to find an update pair with the same index. The update list must be searched in order from the beginning to find the the most recently updated version of the value. This ordering allows any number of modifications to the same index, while maintaining determinacy. Temporal ordering is maintained by the order of the list. We have eliminated copying the array, but now accessing an element requires searching a list, which is expensive; it is still sequentialized since the list must be searched in order. By examining the example code above, it can be seen that each element in the array is only written once. If we make this a requirement on the data structure then the ordered list of update pairs can become an unordered set and the order in which elements are added becomes unimportant; the sequentialization requirement has been removed. Before an element is added to the set, its value is undefined. Since reading requires a defined value,

we require a read to wait until the value it needs is added to the set.

Is determinacy maintained when using unordered sets? Every read of a given index, j, gets the value that is written to index j. This value is unique since it is written only once. Reads and writes can be done in any order and the results are the same, so determinacy is maintained. We have just defined a new method of accessing a data structure. Structures which use this method of access are called "I-structures" because of the *incremental* way in which each write extends the set of update pairs. A more formal definition of I-structures is given in Section 2.2. I-structures can be used to create any type of data structure, not just arrays. Lists can be created by using I-structures for the cons cells. I-structures are sometimes referred to as being data structures, usually arrays, but this is a misnomer. I-structures are actually a method of synchronizing memory accesses. The intended interpretation is a data structure accessed using I-structure synchronization. The confusion arises because conventional memory has only one, non-synchronizing, access method, so no distinction is made between data structures and memory access methods.

It might be possible for a compiler to detect cases where each location in a data structure is written only once and convert these data structures to use I-structures. The detection of "write-once" access patterns is not possible in all cases. By defining a data structure to be an I-structure, the programmer states that the write-once property will hold. If the programmer is wrong, the multiple writes are detected by the hardware at run time.

Now, let us use I-structures to rewrite the example program. The function I_array returns an array of I-structure locations.

```
def fib_array n = { x = I_array (1,n);
                    x[1] = 1;
                    x[2] = 1;
                   in {for i ← 3 to n do
                       x[i] = x[i-1] + x[i-2] };
                      finally x}};
```

This time, only one copy of the array is created. There is no copying and no extra sequentialization. In fact, in Id, all data structures (i.e., arrays, tuples, lists, closures, etc.) are internally represented using I-structures.

Another way to look at I-structures is to view each location as a dynamic arc in a

dataflow graph. See Figure 2.1. Each arc has a source, the instruction that writes the location, and possibly many destinations, the instructions that read the location. The arcs are dynamic because at run time each instruction chooses the index of the location it will access. If the source and destinations of each arc are known at compile time, the I-structure storage can be eliminated and replaced with direct arcs. Here it is clear why each I-structure location may be written at most once; each dataflow arc can have only *one* source.

For a hardware design, we need only those properties that must be maintained by the hardware in order for I-structures, as seen by the language, to work properly. The hardware should also provide an efficient implementation. These issues are covered in the following sections, along with other synchronization methods.

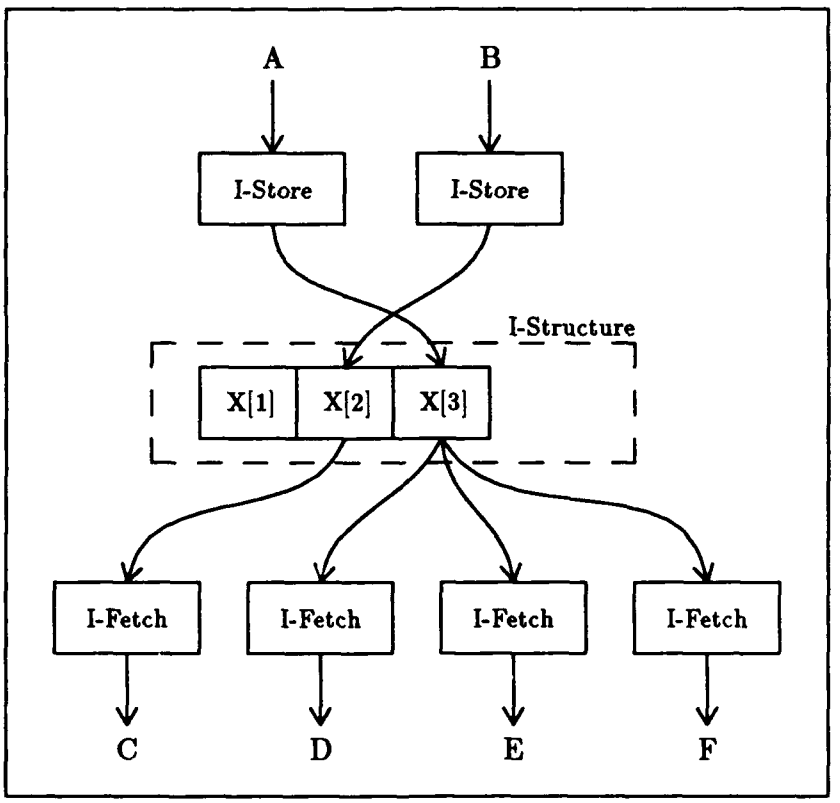


Figure 2.1: Illustration of I-structure cells acting as dynamic dataflow arcs.

2.2 What are I-Structures?

I-structures are a method of memory synchronization that is invariant to the temporal ordering of operations. In other words, the same results are returned independent of the order in which reads and writes are performed. The order may change, but the value returned to each read request will be remain invariant. Temporal invariance is important in parallel computers because it reduces the synchronization required between processors. Each processor reads the values it needs when it wants them; synchronization is handled by the memory. The time between a read request and the return of a value is not fixed; it could be a long time. Operations must be split-phase to tolerate this potentially long latency of read operations. Writes also have a potentially long latency, but, since no value is returned, the processor does not need to wait for a write to reach the memory. The latency of the write is due only to the latency of the network; writes are not delayed because of synchronization.

How is the time invariance property maintained? Some restrictions are needed since general reads and writes do not have this property. Consider, for example, the following code segments running on a two-processor, shared-memory computer. Each program is running in parallel on a separate processor.

Processor A		Processor B	
A1:	LOAD R1 ← [1000]	B1:	LOAD R2 ← [1002]
A2:	INC R1	B2:	INC R2
A3:	STORE R1 → [1002]	B3:	STORE R2→ [1004]

The result stored in location 1004 will depend on the order in which instructions A3 and B1 are executed. Given an initial memory configuration:

Address	Value
1000:	1
1002:	0
1004:	0

If instruction A3 is executed before B1 then the final value of location 1004 will be 3, the desired result. If B1 is executed before A3 then the value will be 2. The problem in this

program is that code B can read location 1002 before code A writes the value that B needs. What we need to do is incorporate the knowledge that code B needs the value written by code A into the memory system.

Each I-structure memory word is initially undefined. Its value is defined by a write. A read requests the defined value of a location. If an undefined location is read, the read request is deferred until the location is defined by a write. At that time, the value is sent to the deferred request. The requester sees this simply as a read that takes a long time.

By disallowing redefining the value of locations, the memory, as a whole, monotonically approaches a unique final state. Because the final state is unique, the order in which the changes are made is not important; no information can be lost. This is key to maintaining the temporal invariance of I-structures. Each write adds to the definition of the memory. To prevent redefinition of a location, each I-structure location can be written at most once. This is the most restrictive property of I-structures and the major difference from conventional memory, in which every location has an initial value. A location in conventional memory is redefined on every write to that location.

I-structures can be used to correct the problem in the example program presented above. The memory locations are replaced by I-structures with the following initial values:

Address	Value
1000:	1
1002:	*Undefined*
1004:	*Undefined*

Instruction A3 defines the value of location 1002. If instruction B1 executes before A3 then its read request will be deferred until A3 writes the value 2 into location 1002; only then is the value returned to B1. This stalls the execution of code B until the value it needs in location 1002 is defined. If B1 executes after A3, it will immediately get the value and not notice the I-structure synchronization. The only possible final value for location 1004 is now 3. This type of situation is referred to as a producer/consumer relationship; code A produces a value, at location 1002, that code B consumes.

I-structures are not a data structure, but a method of memory access with which data structures can be constructed. I-structures can be used for arrays, lists, or any other data

structure, even single words. Programmers can view I-structure memory in a multiprocessor system exactly like conventional memory in a uniprocessor, except that it can be written only once.

2.2.1 How I-Structures are Implemented

How does the I-structure memory know if a location has been written? With each word of I-structure memory are associated status bits, referred to as "presence bits." These presence bits are used to store the "state" of each word (i.e., whether it is defined, called *present*). On every read, the presence bits are tested to see if the data requested is available. If the data is not present then the read request is deferred until the value is written. A read request is deferred by storing the request until the location is written. Since many requests may be deferred on the same location, a list of deferred requests is kept for each deferred location. When the data is written into the memory location, a copy of the data is sent to each read request that was deferred before the location was written, and the presence bits are set to *present*. Conventional memories, without presence bits, simply return the value of the memory location at the time the read request arrives, even if that value is undefined; the memory system has no way of knowing the state of a location

A read of a location already written (presence state is *present*) is satisfied immediately, exactly like conventional memories. If a processor requests to read a location that is never written, no value will ever be returned to the requester.

2.2.2 An Analogy

An analogy can be made between an I-structure memory and a mail order bookstore. Each book title is the address of a memory location. The book represents the value stored in the location. To order a book (read a memory location), you mail your address to the bookstore. If the book is available, you are immediately sent a copy of the book. If it is a new book, advertised but not yet in the store, then your name is put on a waiting list; your request has been deferred. When the book comes in, the store mails you, and everyone else on the waiting list, a copy of the book. Separate waiting lists are kept for each title. You are not informed if you are put on a waiting list. The waiting list avoids the need to call every day to check if a book has arrived.

Two different books with the same title would cause confusion, so this is disallowed (each book is written at most once). After a book is not needed any more, its shelf space can be cleared and used to hold a new book. This corresponds to deallocating and reallocating I-structure locations before they are used again. The difficulty in reusing shelf space is determining if the book to be removed from the shelves will be wanted by anyone in the future. A book cannot be removed from the shelves unless it will never be requested again.

2.2.3 Code for General I-Structures

As another way to illustrate the functioning of the I-structure controller, Figure 2.2 presents code in ID that simulates the operations performed by the hardware for each type of request. This is not code that is run on a processor. It illustrates the response of the I-structure controller to requests. Each function represents the operations that are performed when a request of that type is received. The code for the I-FETCH and I-STORE operations is given in Figure 2.2. The arguments to the function are the values included in the request. For a read-like, request the arguments are an address local to the I-structure controller and a continuation for the process sending the request. The controller uses the continuation when sending values to the requesting process.

The local memory of the I-structure controller is represented as an array M of locations with presence bits. The presence bits are encoded as ID algebraic types. Each disjunct of the type is a presence bit state with its associated value for the memory word.

Some explanation of ID syntax is in order. The ":=" operator is the updatable assignment operation; the value on the right-hand side is stored into the location on the left-hand side. This is a conventional, non-synchronizing, write operation. The ":" is the infix operator for cons; it is used to add the left-hand argument to the list on the right-hand side.

The *send* function takes a continuation, c, and a value, v, and starts the execution of the code pointed to by c, giving it the value v. This can be thought of as a message that is sent from the I-structure controller to a processor.

```
% ID code simulating the operations performed for I-Fetch, I-Store and I_Clear.

type I_structure_location = present value | empty | error value
                          | deferred (list continuation);

typeof I_Fetch_op = address → continuation → ();
typeof I_Store_op = address → value → ();
typeof I_Clear_op = address → ();

def I_Fetch_op addr c =
    {case M[addr] of
          present v = {send c v}
     |    empty = { M[addr] := deferred c:nil}
     |    deferred clist = { M[addr] := deferred c:clist}};

def I_Store_op addr v =
    {case M[addr] of
          present v' = {send Errorr_continuation v;
                        M[addr] := error v'}
     |    empty = { M[addr] := present v}
     |    deferred clist = {Satisfy_def_list clist v;
                            M[addr] := present v}};

% Internal function
def Satisfy_def_list clist v = {for c ← clist do
                                  send c v};

def I_Clear_op addr = { M[addr] := empty };
```

Figure 2.2: Simulation of operations performed on the I-structure controller to support I-structures. Generic (Lisp style) lists are used for deferred lists.

2.3 Non-Busy-Waiting Locks

There are cases where the write-once restriction of I-structures is too limiting, such as when the value of a memory location must be changed, but its name, the address of the location cannot be changed. This is the case when a global free list is used for memory allocation. The location that stores the pointer to the head of the free list must be stored in a global location, the address of which is fixed. The value of the location is modified whenever a value is removed from the list. The order in which memory is allocated from the list does not affect the determinacy of the program, but how the values are removed is important. As shown in the example on page 23, simply allowing memory values to be changed results in unpredictable behavior in parallel programs. Errors result when more than one modification to the list overlap in time. When removing an element from the list, there is a period of time during which the free-list pointer is invalid; it still points to the area being allocated and not to the next free area. If the modifications are serialized, everything works. In ID serialization is done with Managers [3].

Can we apply some of the ideas from I-structures to the problem of serialization? In I-structures, reading an undefined value is prevented by the presence bit state, which causes the request to be deferred. Here, we need to prevent reading the free-list pointer while it is invalid (i.e., being modified by someone else). To do this, the free-list pointer location is made to look "*empty*" when it is not valid. This has the same interpretation as it did in I-structures; it says "the location does not contain a (valid) value." I-structure locations are initialized as *empty*; with locks the state is changed from *present* to *empty* when the value is read.

The free-list pointer can be thought of as a physical object which is either in the memory or it is elsewhere being modified. It can only be in one place at a time. Reading the free-list pointer "takes" it out of the memory, leaving the memory location empty. Writing it "puts" it back in the memory location. This is the basis for a new memory synchronization method which allows updating the value of a location, but in a controlled manner that prevents inconsistent values.

The new memory access method is referred to as a "Non-Busy-Waiting Lock" because it restricts access to a location. A lock may have only one owner at a time. The owner of the lock, having exclusive ownership, may modify the locked value safely. While the value

is locked, not only are others not allowed to change the value, but they are also prevented from reading it; while locked, the value in the lock location is invalid because it could be changed by its owner. The value of a locked location can be thought of as a physical object. When locked it is in the possession of the owner of the lock. When no one owns the lock, it resides in the memory location. It can only be in one place at a time. The only way to read or write the value is to obtain ownership of the lock. This prevents inconsistent values from being read while the locked value is being modified. Access to the lock is serialized. The same synchronization primitives used to implement I-structures also support the Lock synchronization at no extra cost.

A request to obtain the value, and ownership, of a lock is made by using a fetch-like operation called "TAKE," also called Read-and-Lock in [14]. The term "TAKE" originates from the view of locked values as physical objects. Taking a lock that is present in memory (i.e., *unlocked*) results in ownership of the lock by the requester. The lock is removed from memory and the value is sent to the requester; the lock location is then *empty* (i.e., *locked*). Receiving the value of a lock implies exclusive ownership of the lock; the value can then be safely modified. The only valid copy of the lock's value is in the possession of its owner.

An attempt to TAKE a lock that is not present in memory (i.e., *locked*) is deferred just as an I-FETCH request of an unwritten location is deferred. The TAKE request is stored on a deferred list to await the return on the lock's value; the presence bits are set to *deferred*.

The owner of the lock releases its ownership by sending the value back to the memory location in a store-like operation called "PUT," also called Write-and-Unlock. Again, the term "PUT" comes from viewing the lock as a physical object. If the location into which the value is PUT has deferred TAKE requests pending, then the value is sent to *one* of the deferred requests; the location remains locked. If there are no deferred requests then the value is written into the memory location and the presence bits are set to *unlocked*.

The major difference between I-structures and Non-Busy-Waiting Locks is the behavior of PUT as compared to I-STORE. The value of an I-STORE is sent to *all* deferred I-FETCH requests *and* written into the memory location. In contrast, PUT sends the value to *only one* deferred TAKE *or, if there are no deferred requests,* writes the value into the memory location. The other TAKE requests on the deferred list are not given the value and, hence, remain deferred.

In this style of locking, the lock and locked value are combined in one location. The presence bits are used to store the state of the lock and the memory word stores the lock's value. Other type of locks, such as semaphores, use separate locations for the lock and value. Acquiring a semaphore lock requires repeatedly requesting the lock until it is obtained. Once the semaphore lock is obtained, the locked value is read from a another location. With Non-Busy-Waiting locks, TAKE combines acquiring the lock and reading the locked value into one operation. The PUT instruction combines the process of writing the new value and releasing the lock, thus the number of operations required to access a lock is reduced, compared to traditional semaphores. More importantly, the need for busy-waiting on locks is removed by deferring the requests to TAKE a lock that is unavailable; only one request is issued. The lock is given to a waiting request as soon as it is released (i.e., PUT back in memory). The requester does not need to poll the lock waiting for it to be released. This saves the processor time and reduces the latency of acquiring the lock.

An example will help to illustrate how locks work: consider executing the two code segments A and B, given below, in parallel on two different processors with shared memory. The goal is for each processor to get a unique number, possibly to decide which part of a shared task each will perform. Which processor gets which number is not important, but they must not both get the same number. First, we will try it without using locks.

Processor A	Processor B
A1: LOAD R1 ← [1000]	B1: LOAD R2 ← [1000]
A2: INC R1	B2: INC R2
A3: STORE R1 → [1000]	B3: STORE R2→ [1000]

Initially the value of location 1000 is 0. If all of code segment A is executed before code segment B then registers R1 and R2 will contain the values 1 and 2 respectively. If, instead, B executes before A, then registers R1 and R2 will contain 2 and 1 respectively. Either of these outcomes is correct.

A problem arises when execution of the two code segments is intermixed or they are executed simultaneously. If instructions A1 and B1 are executed before either A3 or B3, then both registers will load the value 0 and both will end up containing the value 1, an incorrect result. A mechanism is needed to sequentialize the execution of the two code

segments. This can be done by replacing the LOAD and STORE instructions with TAKE and PUT instructions, to give the following two new code segments:

Processor A		Processor B	
A1:	TAKE R1 ← [1000]	B1:	TAKE R2 ← [1000]
A2:	INC R1	B2:	INC R2
A3:	PUT R1 → [1000]	B3:	PUT R2 → [1000]

Now, let us reexamine the problematic execution sequence, (A1, B1, ... A3, B3). Again, location 1000 initially contains the value 0. Instruction A1 TAKEs the value 0 from location 1000, leaving it *locked* (*empty*). Instruction B1's TAKE request is deferred on location 1000 because the location is *locked*; the location becomes *deferred*. Code segment A continues to execute, using the value 0, until A3 PUTs the value 1 back into location 1000. Since location 1000 is *deferred* with the TAKE request from instruction B1, the value 1 is sent to instruction B1 instead of being written to memory. Code segment B then completes execution and PUTs the value 2 into location 1000; Since no TAKEs are deferred on the location, the value 2 is written into memory. By stalling the completion of instruction B1, the order of execution appears to be A1, A2, A3, B1, B2, B3 which was stated to be correct previously. It is important to note that the use of split-phase transactions actually allows instruction B1 to complete execution on the processor, which issues the TAKE request to the memory system; it is the result of the TAKE request which is stalled by the memory.

2.3.1 Extending the Analogy to Locks

To extend the book analogy to Non-Busy-Waiting locks, the bookstore is replaced by a lending library. If you want to borrow a book, you write to the library and ask them to send you the book. If the book you want is out on loan, the library adds your name and address to a waiting list (it is assumed the library has only one copy of each book). If the book is available, the library checks it out and sends it to you. You are allowed to re-write a book in your possession (this is a progressive library). When you are done with the book, you return it to the library so that others can check it out and read your changes. When a book is returned, the library either sends it to *one* of the people on the waiting list or puts it back in the stacks. The contents of the book when you get it can depend on the order in which other people have had the book.

There is a fairness issue in choosing the next person from the waiting list. It is possible that some people will wait much longer than others. If there is a finite supply of requests, everyone who wishes will eventually get the book. When the number of requests could be unbounded, or fairness is important, the waiting list should be made first-come first-serve; our implementation is last-in first-out.

2.3.2 Multiple Value Locks

The locks presented so far allow only one owner of the lock at any given time, these are simply called locks. The lock mechanism can be extended to allow multiple simultaneous owners of the lock. These locks are called multiple-value locks. Multiple values are stored in the lock location; each value may have a different owner. The multiple values are stored on a linked list, similar, but not always identical, to deferred read lists. The multiple values of the lock need not be the same. Each request to TAKE the lock removes one value from the list, until none remain; further TAKE requests are then deferred. When the lock location holds deferred TAKEs, PUTs work exactly the same as for single-locks; *one* deferred TAKE is satisfied by each PUT. A PUT to a location that is already unlocked (i.e., values are *present*) adds its value to the linked list of other values. The list of multiple values is stored like a deferred list when the lists are stored locally, except that the list contains values instead of return tags. Distributed deferred lists are a special case and can not be extended to hold multiple values. See Section 3.8.

Multiple value locks can be viewed as a rendezvous point for two streams, one a stream of requests and the other a stream of values. As they arrive, each value is paired with a request, thus satisfying the request. Any value may be paired with any request, pairing is done by first availability. The first to arrive, from either stream, when the location is *empty* is stored in the location, to await an arrival on the other stream. If a request arrives when a value, or list of values, is waiting in the location then one of the values is removed from the location to satisfy the request. If the new arrival is of the same type as those already waiting, it is added to the list. At any given time, there can be either a list of waiting requests or a list of waiting values, but not both. In order for a value to be put on a list, there must not be any requests waiting, so the list of waiting requests must be exhausted before a list of waiting values is created. For this reason, only one list is needed, since there

will only be one type of list at any given time.

2.3.3 An Analogy for Multiple Value Locks

A taxi stand is a synchronization place for taxis and customers. The taxis represent the values of a multiple-lock, and customers represent the requests to TAKE the lock. The system can be in one of three states: a queue of waiting taxis, a queue of waiting customers, or both queues empty. Customers arriving to a queue of waiting taxis each take one taxi from the queue. When the taxi queue is empty, customers will form a queue. Taxis arriving to a queue of waiting customers will take one customer from the queue and depart again immediately. A queue of customers and taxis will never exist at the same time; the waiting customers would use the available taxis. Of course, in the real world, queues of both customers and taxis are often experienced (often with great annoyance to the observer waiting in the line) because it takes time to get into a taxi; this does not occur in our model. As with the values of a multiple-lock, the number of taxis in the system is a constant. The number of customers (requests) can be unlimited.

2.3.4 Code for Simulating Lock Operations

Figure 2.3 shows ID code for simulating the behavior of the I-structure controller in response to TAKE and PUT requests. This code, which does not handle multiple-value locks, is very similar to that for I-FETCH and I-STORE. The only difference between the code from I-FETCH for TAKE is the addition of the presence state change from *present* to *empty*. An explanation of ID syntax and special functions included for the simulation are included with the code in Section 2.2.3.

2.3.5 Reading a Lock

It might seem reasonable to use an I-FETCH to look at the value of a lock without acquiring ownership, but it won't work. The hardware determines the type of the location by the type of request used to access the location. If I-FETCH is used on a lock location, the hardware treats the location as an I-structure location. This will work correctly in the *present* state, but not if the request is deferred. A PUT to the location will treat all deferred request as

```
% ID code simulating the operation of Take, Put and Initialize-Lock.

type Lock_location = unlocked value | locked | error value
                   | deferred (list continuation);

typeof Take_op = address → continuation → ();
typeof Put_op = address → value → ();
typeof Initialize_Lock_op = address → ();

def Take_op addr c =
    {case M[addr] of
          unlocked  v = { send c v;
                          M[addr] := locked }
     |    locked  = { M[addr] := deferred c:nil}
     |    deferred clist = { M[addr] := deferred c:clist}};

def Put_op addr v =
    {case M[addr] of
          unlocked  v' = { send Error_continuation v;
                           M[addr] := error v'}
     |    locked  = { M[addr] := unlocked  v}
     |    deferred c:clist = { send c v;
                               M[addr] := if clist == nil
                                          then locked
                                          else deferred clist}};

def Initialize_lock_op addr = { M[addr] := locked };
```

Figure 2.3: Simulation of operations performed on the I-structure controller to support Locks Generic (Lisp style) lists are used for deferred lists.

deferred TAKE requests. The proper way to read the value of a locked location is to TAKE it, look at it, and then PUT it back; the value of a lock is only valid for its owner. This EXAMINE operation can be optimized to make it act more like an I-FETCH, but it is a lock type operation,

2.4 Delayed Triggers (for Lazy Data Structures)

Delayed triggers are used to implement lazy data structures [6] in which a value is not computed until it is requested. Consider filling an array, x, with the values of a function, say f, where the value at each index depends on the index (i.e., x[i] = (f i)). The Fibonacci code on page 21 is an example of such an operation. The code executes a loop which executes function f for each index and stores the value into the array and then the values are read. Since I-structures are used for the array, the reads can proceed writing the value; they are deferred until the value is written.

Now, consider the case where only a few of the array locations are ever read, and the function f is expensive to compute. Calculating the values that are not read is wasted computation. It would be better to only compute the values that are needed, but those aren't known ahead of time. Instead of creating the array and then reading the values from it, the function f could be called directly for each index needed. Reading an index multiple times causes the value to be recomputed each time. The original method of filling the array only calculated the value for each index once. Lazy data structures combine the advantages of only calculating needed values and then remembering the value.

Instead of executing the function f, everything needed to execute the function, its environment (free variables), arguments (i) and a pointer to the function's code, are stored in a data structure, called a Thunk in [6]. A continuation is stored into the array index which, when released, uses the data in the Thunk to execute f and store the computed value in the array; the continuation is referred to as a Delayed Trigger. The array location is marked as *delayed*. A request to fetch a location that is delayed releases the delayed trigger continuation to start the computation of the requested value; the fetch request is deferred just as if the location had been *empty*. The deferred continuation replaces the delayed trigger continuation in the I-structure location. Other fetch requests for the same location are also

deferred, as just as they are for I-structures.

Delayed triggers are completely invisible to the requester, since a location containing a delayed trigger (presence bits in the *delayed* state) acts exactly like an *empty* location as far as the requester is concerned. The only difference, the release of the delayed trigger, is invisible to the requester. A *delayed* location is set up by a special STORE-DELAY instruction that stores the delayed trigger and sets the presence bits to *delayed*. Delayed triggers can only be stored in *empty* locations since there must be somewhere to store the trigger's continuation.

Delayed triggers are used to generate an event the first time the location is accessed by any method (I-FETCH, I-STORE, TAKE, or PUT). The event is signaled by returning a value to the process that stored the delayed trigger. The value returned is unimportant since it is returned only to pass along the event. For lazy evaluation, the event is used to start a computation. Delayed triggers can also be used to set breakpoints on locations for debugging or other purposes. For example, in the Monsoon prototype, delayed triggers are used to indicate when the activation frame free list has been exhausted. A delayed trigger is stored in the last location of the free list. Any access of the last location is an attempt to exceed the available free activation frames. This signals an error condition which halts the processor. A breakpoint need not halt the processor; it could simply record the event or report it to the user.

2.4.1 Delayed Trigger Analogy

In the book analogy, a delayed trigger is a watch placed on the title of a book which has not yet been written. The first time someone tries to buy or borrow the book, the watch is triggered, and whoever ever placed the watch is notified. The person asking for the book is placed on a waiting list, since the book is not available; they do not find out about the watch, which is removed.

This can be taken advantage of by a "lazy" publisher. The publisher advertises the titles of books it might write, places watches on those titles with bookstores and waits. The publisher is notified by the watch when someone requests one of the non-existent books. Only after someone has ordered a book is a writer commissioned to write it. In this analogy, people on the waiting list are committed to buy the book, no matter how long they may

wait for it to be written. In reality people tend to cancel orders that aren't fulfilled since they suspect that a tactic like this is being used by the seller. Subsequent orders for the same book are added to the waiting list; the watch was removed by the first request and has no effect on these requests. The buyer cannot distinguish a "lazy" publisher from a very slow mail system.

In this system, only one watch can be active on each title. Placing a second watch causes notification of the first watch, just as if someone ordered the book; the second watch then replaces the first as the active watch. A watch is triggered by *any* activity related to the book's title.

2.4.2 Code to extend I-Structures and Locks with Delayed Triggers

The following code is an extension to the code presented in Figure 2.2. One new presence bit state, *delayed*, and the code to simulate it are added. The extensions made to I-FETCH and I-STORE are made to the code for TAKE and PUT in Figure 2.3. The delayed trigger is a continuation which is stored in the location using the STORE-DELAY instruction. Code to simulate the STORE-DELAY instruction is included in Section 2.4.3.

```
% ID code for extending the simulation of I-Structures
% to include Delayed Triggers.

type I_structure_location = present value | empty | delayed continuation
                          | error value | deferred (list continuation);

def I_Fetch_op addr c =
    {case M[addr] of
          ⋮
     |    delayed trigger = { send trigger trigger;
                             M[addr] := deferred c:nil}

def I_Store_op addr v =
    {case M[addr] of
          ⋮
     |    delayed trigger = { send trigger trigger;
                             M[addr] := present v}
```

2.4.3 Code for Storing a Delayed Trigger

The following code simulates the operation used to store a Delayed Trigger into an I-structure or Lock location. The trigger is a continuation, *c*, of the process that will compute the value for the delayed location. The trigger is only stored if the location is *empty*, otherwise the trigger is released to signal that the location has already been accessed.

% Simulation code for operation to store a Delayed Trigger.

typeof Store_Delay_op = address → continuation → ();

def Store_Delay_op addr trigger =
    {**case** M[addr] **of**
        present v = {send trigger v}
    | *empty* = { M[addr] := *delayed* trigger}
    | *deferred* clist = {send trigger clist}};

2.5 Imperative Non-Synchronizing Memory Operations

The non-synchronizing memory accesses used on conventional computers are referred to here as imperative operations. Imperative operations are simple to implement because they do not use the presence bits. The operations are performed as they are received; imperative operations are never deferred. Imperative operations are useful on dataflow computers for routines that need complete control and can guarantee the correct ordering of operations, such as system level routines. Conventional processors could be supported by an I-structure memory using just these operations. The ability to support all of the functionality of a conventional, non-synchronizing, memory demonstrates that the synchronizing memory is more powerful.

Memories that only support Imperative operations are simpler and cheaper to design and build, which explains why they are used in current computer designs. The synchronization mechanisms require extra hardware and memory, but the functions they provide are necessary for supporting determinacy in languages such as ID.

2.5.1 Code to Simulate Imperative Operations

The following code illustrates the operation of Imperative operations. The presence bit state of the memory location is ignored for imperative operations; this is represented by the type *any-state*. A more complete version of how the I-structure controller implements these operations is given in the code at the end of the chapter.

```
% ID code simulating the operation of Imperative Read and Write.

type Imperative_Location = any-state v;

def Read_op addr c = {case M[addr] of
                        any-state v = {send c v}};

def Write_op addr v = { M[addr] := any-state v};
```

2.6 Unifying Presence States

Each of the synchronization methods presented in this chapter has its own definition of the presence bit states. These are represented in the ID simulation code as different algebraic types. These types are unified to allow the hardware to use the same presence bits for all of the methods. Two of the lock states are redefined:

$$locked = empty$$

$$unlocked = present$$

The imperative operations do not change the presence state, so those functions simply perform the same operation for every state. The state *any-state* maps to the current state of the presence bits. The unified ID type is given below.

```
type Memory_location = present value | empty | error value
                     | delayed continuation | deferred (list continuation);
```

Since each word has presence bits, the type of memory access can be chosen on a per-word basis. The hardware does not record the type of access with a location; the access

method is determined by the operations that are performed on the location. If I-FETCH and I-STORE are used, then the location acts as an I-structure. Using TAKE and PUT defines a Lock. The result of mixing different types of accesses on a single location is not defined.

The following code segments are a complete collection of all the code segment defined previously in this chapter. The functions do not return any values directly. All values are returned using the *send* function. The type signature for the functions are given at the beginning of the code below.

```
typeof I_Fetch_op, Take_op, Read_op = address → continuation → ();
typeof I_Store_op, Put_op, Write_op = address → value → ();
typeof I_Clear_op, Initialize_lock_op = address → ();
typeof Store_Delay_op = address → continuation → ();

% ID code simulating the operations performed for I-Fetch, I-Store and I_Clear.

def I_Fetch_op addr c =
    {case M[addr] of
          present v = {send c v}
     |    empty = { M[addr] := deferred c:nil}
     |    error v = {send Error_continuation c}
     |    delayed trigger = { send trigger trigger;
                              M[addr] := deferred c:nil}
     |    deferred clist = { M[addr] := deferred c:clist}};

def I_Store_op addr v =
    {case M[addr] of
          present v' = {send Errorr_continuation v;
                        M[addr] := error v'}
     |    empty = { M[addr] := present v}
     |    error v' = {send Error_continuation v}
     |    delayed trigger = { send trigger trigger;
                              M[addr] := present v}
     |    deferred clist = {Satisfy_def_list clist v;
                            M[addr] := present v}};

% Internal function
def Satisfy_def_list clist v = {for c ← clist do
                                send c v};

def I_Clear_op addr = { M[addr] := empty };
```

%%% *This code is continued on page 41.*

%%% *This code is continued from page 40.*

% ID *code simulating the operation of Take, Put and Initialize-Lock.*

```
def Take_op addr c =
    {case M[addr] of
          present  v = { send c v;
                         M[addr] := empty }
     |    empty  = { M[addr] := deferred c:nil}
     |    error v = {send Error_continuation c}
     |    delayed trigger = { send trigger trigger;
                              M[addr] := deferred c:nil}
     |    deferred clist = { M[addr] := deferred c:clist}};

def Put_op addr v =
    {case M[addr] of
          present  v' = { send Error_continuation v;
                          M[addr] := error v'}
     |    empty  = { M[addr] := present  v}
     |    error v' = {send Error_continuation v}
     |    delayed trigger = { send trigger trigger;
                              M[addr] := empty v}
     |    deferred c:clist = { send c v;
                               M[addr] := if clist == nil
                                          then empty
                                          else deferred clist}};

def Initialize_lock_op addr = { M[addr] := empty };

% Simulation code for operation to store a Delayed Trigger.

def Store_Delay_op addr trigger =
    {case M[addr] of
          present v = {send trigger v}
     |    empty = { M[addr] := delayed trigger}
     |    error v = {send Error_continuation trigger}
     |    delayed c = { send Error_continuation trigger;
                        M[addr] := error c}
     |    deferred clist = {send trigger clist}};
```

%%% *This code is continued on page 42.*

%%% *This code is continued from page 41.*

% ID *code simulating the operation of Imperative Read and Write.*

```
def Read_op addr c = {case M[addr] of
                        present foo = {send c foo}
                      | empty foo = {send c foo}
                      | delayed foo = {send c foo}
                      | error foo = {send c foo}
                      | deferred foo = {send c foo}};

def Write_op addr v = {case M[addr] of
                         present foo = { M[addr] := present v}
                       | empty foo = { M[addr] := empty v}
                       | delayed foo = { M[addr] := delayed v}
                       | error foo = { M[addr] := error v}
                       | deferred foo = { M[addr] := deferred v}};
```

Chapter 3

Deferred List Implementations

Two extensions are made to a conventional memory to implement the synchronization methods described in Chapter 2, presence bits and deferred lists. Presence bits are added to each word of memory and are used to encode the state of the memory word. The presence bits are exactly like those used in the Monsoon processor [13]. The presence bits allow the I-structure controller to record the order in which fetch and store requests arrive and change its action based on the state of the location requested. The order in which requests arrive determines the state of the presence bits. Conventional memories do not need presence bits, since the order of arrival does not affect the operations performed.

The most important feature of the I-structure controller is the ability to defer processing of a request until a later event. This is used to effectively reorder operations to produce correct results. When an I-FETCH arrives at a location before the I-STORE, the I-FETCH is delayed until the I-STORE arrives. How operations are deferred is of great practical importance to the design of an I-structure controller. Since the operations are deferred until a later event on the same location, the deferred operations should be associated with that location. A list is formed at each location on which requests are deferred. The list stores the deferred requests. I-structure semantics do not limit the number of operations that may be deferred on a location; a deferred list can grow to any length. Linked lists are used for the deferred lists because they are simple and extendable.

Four methods of storing deferred lists are presented. Two methods store the lists locally on the controller, one using two word cons cells (Section 3.5) and the other adds an extra Link field to each memory word (Section 3.6). The other two methods distribute the storage

of the linked lists across the processors (Section 3.9 and Section 3.10).

3.1 Terminology

In the following discussion, requests are referred to as R_i, indicating that it is the i^{th} request to arrive at the requested location on the I-structure controller. R_1 is the first request to arrive. The process that issues request R_i is called P_i. With each I-FETCH or TAKE request the process sends a continuation that is used to return data to it. A continuation is like the "process status word" and is a pointer to the code that is executed when the requested value is returned. In machines having different contexts to allow multiple invocations of the same code, the continuation also includes a context pointer (e.g., a stack pointer). The continuation for process P_i is represented as C_i. The internal encoding of continuations is not important here, except that a continuation must not be larger than a value in order to allow it to be stored as a value.

3.2 Format of Requests

A request, R_i, of the form $<$ I-Fetch A, C_i $>$, requests that the value of the I-structure location at address A be sent to continuation C_i of processes P_i. Requests to store a value are of the form $<$ I-Store A, v $>$, where v is the value to be stored at address A. The format of the response which returns the requested value, v, to the continuation, C_i, of the requesting process, P_i, is $< C_i,\ v >$.

3.3 Storing the First Deferred Request

The first request, R_1, deferred on a location can be stored for free; no extra memory needs to be allocated. The *empty* location is used to store C_1, the continuation of process P_1 which issued request R_1. The presence bits are changed from the *empty* state to the *deferred* state to indicate that the location now contains the continuation of a deferred request. No extra space needs to be allocated to store this deferred list of length one. The differences between the three methods of storing deferred lists are only apparent for lists with more than one

deferred request. All the list methods use this way of storing C_1, the continuation of the first request deferred.

Figure 3.1 illustrates a section of memory after some operations have been performed. The left most column represents the presence bits; "E" for *empty*, "P" for *present* and "D" for *deferred*. The right column is the value field. The first location, A1, is *empty*; the contents of the value field are ignored.

The second location, A2, contains the *present* value of 251. The third, A3, is *deferred*; it holds the continuation, C_1, of the first, and only, request, R_1, which is deferred on that location.

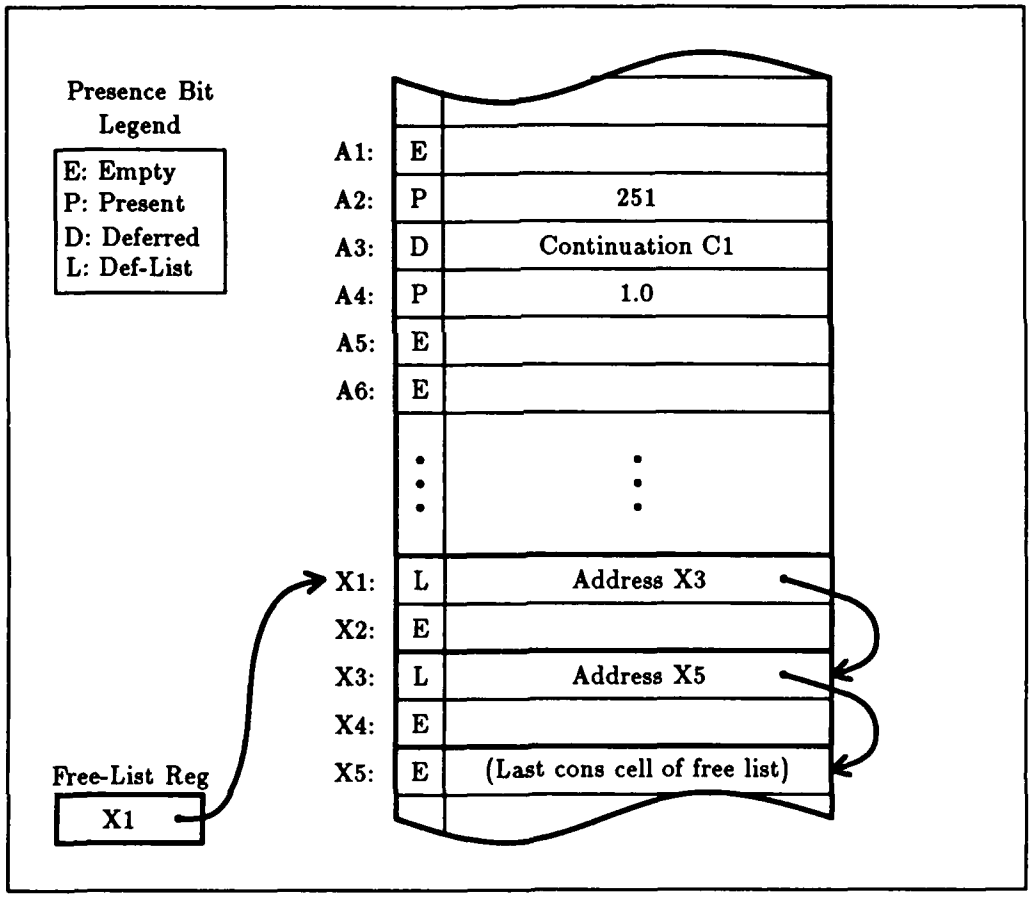


Figure 3.1: Example of I-Structure Memory with one deferred continuation, C_1. Addresses X1 and X2 form a cons cell, as do X3 and X4. Both are on the free-list.

3.4 Local Deferred Lists

Since storing deferred requests is an operation performed by the I-structure controller, the simplest place to store these lists is on the I-structure controller itself. It has local memory in which to store the linked lists. The processor never need deal with I-structures because the I-structure controller handles all of the details of deferring requests. When an I-FETCH request is deferred, the I-structure controller allocates space from its local unused deferred list space to extend the deferred list.

3.4.1 Local Free-List Management

Cons cells are allocated by popping them off of a list of free cons cells. The free-list register contains a pointer to the first cons cell on the free list. When a deferred list is satisfied, all of its cons cells are pushed back onto the free list; no garbage collection is required. Each I-structure controller has its own free-list and associated free-list register.

Before allocating a new cons cell, the presence bits of the location pointed to by the free-list register must be tested to determine if there is any free space available. The *deferred-list* state indicates an available free cons cell. The *empty* state is used to mark the end of the free-list. The I-structure controller halts when it attempts to pop an *empty* free-list location because it has exhausted its free deferred list space. Since it cannot finish processing the current request, this a fatal error. When the I-structure controller halts, it blocks its input from the network, which prevents any communication from the processor that is attempting to solve the problem and which could deadlock the whole network. The only choice is to yell for help from the host processor, if there is one. In the Monsoon system this can be done over the VME bus, but it is slow. The blocked input may cause the whole network to become deadlocked.

3.5 Local Lists using Cons Cells

The first method of storing deferred lists is to use cons cells (two consecutive memory words) to build the linked lists. The first word, called the CAR (a term from Lisp), stores an element of the list, a deferred request's continuation. The second word, called the CDR,

contains a pointer to the next cons cell on the list. A pointer to the head of the deferred list is stored in the *deferred* I-structure location.

In Lisp, the end of a list is signified by a null pointer in the CDR of the last cons cell. In this method, the presence bits of the CDR are used to mark the end of the list allowing the last CDR location to be used to store another deferred continuation. The last cons cell actually contains the continuations of two deferred requests. In Lisp terminology, the last cons cell is a dotted pair. This fits in nicely with the simple model of extending the deferred list explained below and uses one less cons cell of storage than does a Lisp list.

The pointer to the next cons cell on the free-list are stored in the CAR of the cons cells, in the local cons cell method; just the opposite of the deferred lists. This is done to simplify the memory access pattern. In the local Link field method, the free lists pointers are stored in the link fields. In Figure 3.1, location X1 is the CAR and X2 the CDR of a cons cell; X3 and X4 form another cons cell. The free-list register contains the address, X1, of the first cons cell on the free list. Location X1, in the *deferred-list* state, contains the address of X3, the next cons cell on the free-list. In the example, Figure 3.1, there are only two cons cells on the free list. The presence state of *empty* of location X5 indicates that it is the end of the free-list; the state of all other free-list pointer cells is *deferred-list.*

In dynamic memory, an exchange cycle is only slightly longer than either a read or a write cycle since most of the access time is spent setting up the address. For this reason, an exchange is used to replace a read followed by a write to the same location. When accessing consecutive memory words, as in a cons cell, the page-mode feature of dynamic memories can be used to decrease the access time at the cost of more complex control signal timing.

3.5.1 Creating Deferred Lists (Local Cons Cells)

Referring back to the example in Figure 3.1, the continuation, C_1, of the first request deferred, R_1, is stored in the I-structure location it requested (A3) and no extra memory is allocated. The location marked with a "D" in the left column of Figure 3.1 is the *deferred* location. Subsequent reads are deferred by allocating a new cons cell in which to store the new deferred read request's continuation, and pushing the new cons cell onto the beginning of the deferred list.

The second request deferred, R_2, requires a cons cell to extend the deferred list. The free-list register points to the new cons cell, X1. Cons cell X1 is removed from the free list by reading its value, X3, into the free-list register. The new cons cell, X1, is added onto the deferred list by copying the contents, *deferred* C_1, of the I-structure location, A3, into the CDR of the new cons cell, X2. The new request's continuation, C_2, is written into the CAR of the new cons cell, X1, to complete the process. Figure 3.2 illustrates the resulting memory state. By using exchange cycles, the allocation of a new cons cell and its addition to the deferred list are combined so that each memory location is accessed only once. The algorithm used is:

1. Exchange the free-list pointer with the value in the original location, call it W. (W is either a pointer to a free list cons cell, or continuation C_1.
2. Exchange the continuation of the new request with the CAR of the new cons cell (pointed to by the free-list register). Store the value read from the CAR into the free-list register. (If the state of the CAR was *empty* then the free-list is empty; **Halt.**)
3. Store W into the CDR of the new cons cell. The presence state of the CDR is set to the same state as the location from which W was read.

Figure 3.2 illustrates a deferred list containing two deferred requests, R_1 and R_2, with continuations C_1 and C_2, constructed using local cons cells. It is an extension of the example given in Figure 3.1; a second read request, R_2, with continuation C_2, has been added to the deferred list. Figure 3.3 extends the example by adding a third deferred continuation, C_3. The deferred location, A3, contains a pointer to the deferred list, indicated by a presence state of *deferred-list*. Extending the deferred list only involves changing the deferred location and the CAR and CDR of the new cons cell added to the list.

3.5.2 Satisfying Deferred Lists (Local Cons Cells)

Deferred reads are satisfied when a value, V, is stored into the location, A3 in Figure 3.3. The value is sent to the continuation, C_i, of each process, P_i, which made a request that was deferred. If there is only one deferred request, R_1, then the location, A3, contains its continuation, C_1; V is sent to process P_1 using C_1.

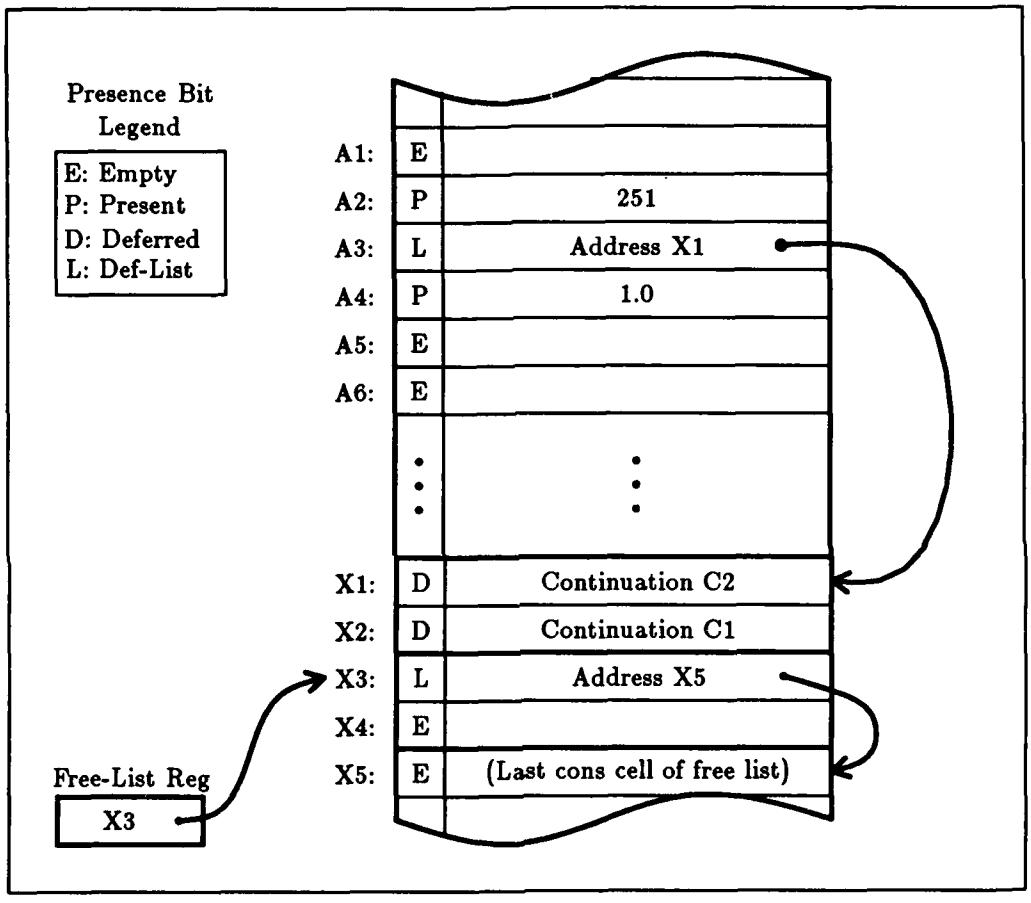


Figure 3.2: Two deferred requests, with continuations C_1 and C_2, deferred on location A3. Local cons lists are used to construct the deferred list which contains one cons cell. The *deferred* state of the CDR indicates the end of the list.

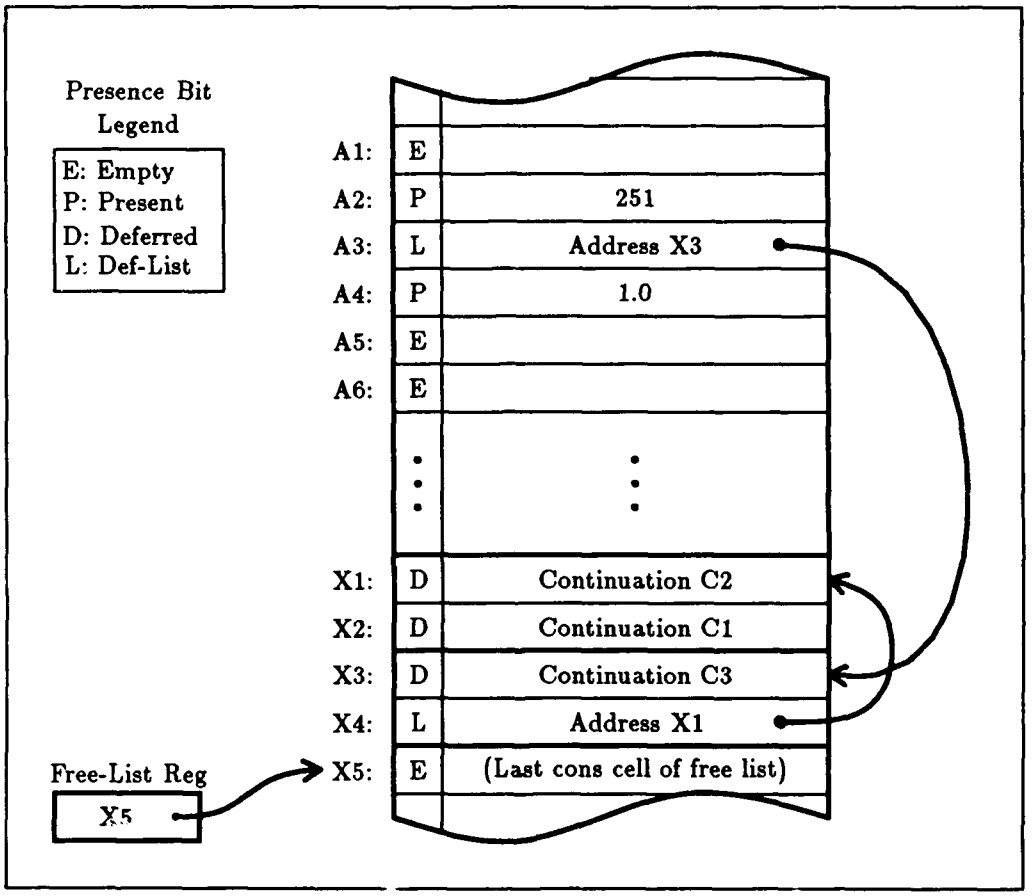


Figure 3.3: Extension of deferred list in Figure 3.2 by a third deferred request, R_3, with continuation C_3. Local cons cells are used to store the deferred lists. The requests were deferred in numerical order. Notice the requests are queued in reverse order.

For a list of multiple deferred requests, the I-STORE's value, V, is exchanged with the pointer to the deferred list contained in the deferred I-structure location. This pointer is saved for later use and used to access the first cons cell on the deferred list. Both words of the cons cell are read. The continuation, C_i, read from the CAR, is used to send the value, V, to process, P_i. If this is not the last cons cell of the list, as indicated by a *deferred-list* presence bit state of the CDR, then the CDR contains a pointer to the next cons cell of the deferred list. Otherwise, it is the last cons cell and the CDR contains another deferred continuation, C_2, to which the value V is also sent. Each successive cons cell on the deferred list is accessed until the last cell is reached.

The free-list pointer, X5, is written into the CAR of the last cons cell of the deferred list. The free-list register is then loaded with a pointer to the beginning of the deferred list (stored previously in a temporary register), thus pushing the whole deferred list onto the free list.

3.5.3 ID Code Simulation of Controller Operations for I-Structure

The ID code in Figure 3.4 simulates the behavior of an I-structure controller using cons cells to store deferred lists locally. The allocation of deferred list cons cells from the free list is handled explicitly by the function *cons*. The first word of the cons cell stores a continuation. The second word contains a pointer to the next cons cell of the deferred list, except in the last cons cell where it stores the last continuation. When the deferred list is satisfied, the whole deferred list is pushed onto the free-list. The code which does the push is in *I_Store* in the *deferred-list* case and *Satisfy_def_list* in the *deferred* case.

3.5.4 Deferred Lists in Sigma-1

The Sigma-1 [7] dataflow processor, built at the Electrotechnical Laboratory in Japan, uses the local cons cell method of storing deferred lists on its Structure Elements (SE). The synchronization method used is called B-structures, which are the same as I-structures. A local controller handles all deferred list operations. The three flag bits (the equivalent of presence bits) are stored in static memory while the data is stored in dynamic memory. Each Structure Element has 256 Kwords with 5 bytes per word. Approximately one quarter of the memory space is allocated for deferred list space. Unused deferred list cons cells are

```
% ID code for I-Fetch and I-Store using Local Cons Cell deferred lists.

type I_structure_location = present value | empty | deferred continuation
                          | deferred-list address | error value;

def I_Fetch_op addr c =
    {case M[addr] of
          present v = {send c v}
     |    empty = {M[addr] := deferred c}
     |    deferred c' = {M[addr] := deferred-list (cons c M[addr])}
     |    deferred-list a = {M[addr] := deferred-list (cons c M[addr])}
     |    delayed trigger = { send trigger c;
                              M[addr] := deferred c}
     |    error v' = {send Error_continuation c}};

def I_Store_op addr v =
    {case M[addr] of
          present v' = { send Error_continuation v;
                         M[addr] := error v'}
     |    empty = { M[addr] := present v}
     |    deferred c = { send c v;
                         M[addr] := present v}
     |    deferred-list a = { Satisfy_def_list M[a] a v &
                              free_ptr := a;
                              M[addr] := present v}
     |    delayed trigger = { send trigger v;
                              M[addr] := present v}
     |    error v' = {send Error_continuation v}};

def Satisfy_def_list (deferred c) a v =
    {send c v;
     {case M[a+1] of
          deferred c' = {send c' v;
                         M[a] := free_ptr}
      |   deferred-list a' = {Satisfy_def_list M[a'] a' v}}};

def cons c word = {p = free_ptr &
                   {case M[free_ptr] of
                         empty = Free-List-Exhausted-Error
                    |    deferred-list a = { free_ptr := a}} &
                   {M[p] := deferred c;
                    M[p+1] := word} &
                   in p};
```

Figure 3.4: ID code: I-Structures with Local Cons Cell Deferred Lists

stored on a free-list, allowing the amount of deferred space to be completely configurable by system software. A prototype with 128 Processing Elements and 128 Structure Elements has been built.

3.6 Local Lists Using Link Pointer Field

Since deferred lists are local to each I-structure controller, a local pointer can be used for the CDR pointer. In the local cons cell method, a full memory word is used to store the CDR. The current design has an address space of 16 Megawords per I-structure controller which requires 24 bits to address. So, only 24 bits are needed for a local pointer compared to the 72 bits for the type and value fields of a full memory word. An alternative to two-word cons cells is to add a local pointer field, called the link field, to each memory word. See Figure 3.5 The link field plays the roll of the CDR while the value field is the CAR. Only one memory word, extended with a link field, is required per list element.

The presence bits are used to mark the end of the deferred list. A word in the *deferred-list* state has a deferred continuation in the value field and a pointer to the next word in the Link field. A word in the *deferred* state contains only a deferred continuation in the value field; the link field is ignored. All of the memory words in a deferred list are in the *deferred-list* state, except the last which is in the *deferred* state.

Figure 3.6 shows an example of the same two element deferred list shown in Figure 3.2, constructed here using the link fields. One less word of storage is required, but every word is 24 bits longer. Figure 3.7 shows the same example after a third request, R_3, is deferred. The same example using local cons cell lists is shown in Figure 3.3.

3.6.1 Creating Deferred Lists (Link Pointer Field)

The continuation, C_1, of the first I-FETCH request deferred is stored in the I-structure location on which it was deferred, as in all the other methods; the link field is not used. The resulting memory state is shown in Figure 3.5. Subsequent requests are deferred by allocating a new cons cell word and copying all of the fields into it. The continuation of the new request and a pointer to the new cons cell word are stored in the value and link fields of the I-structure location.

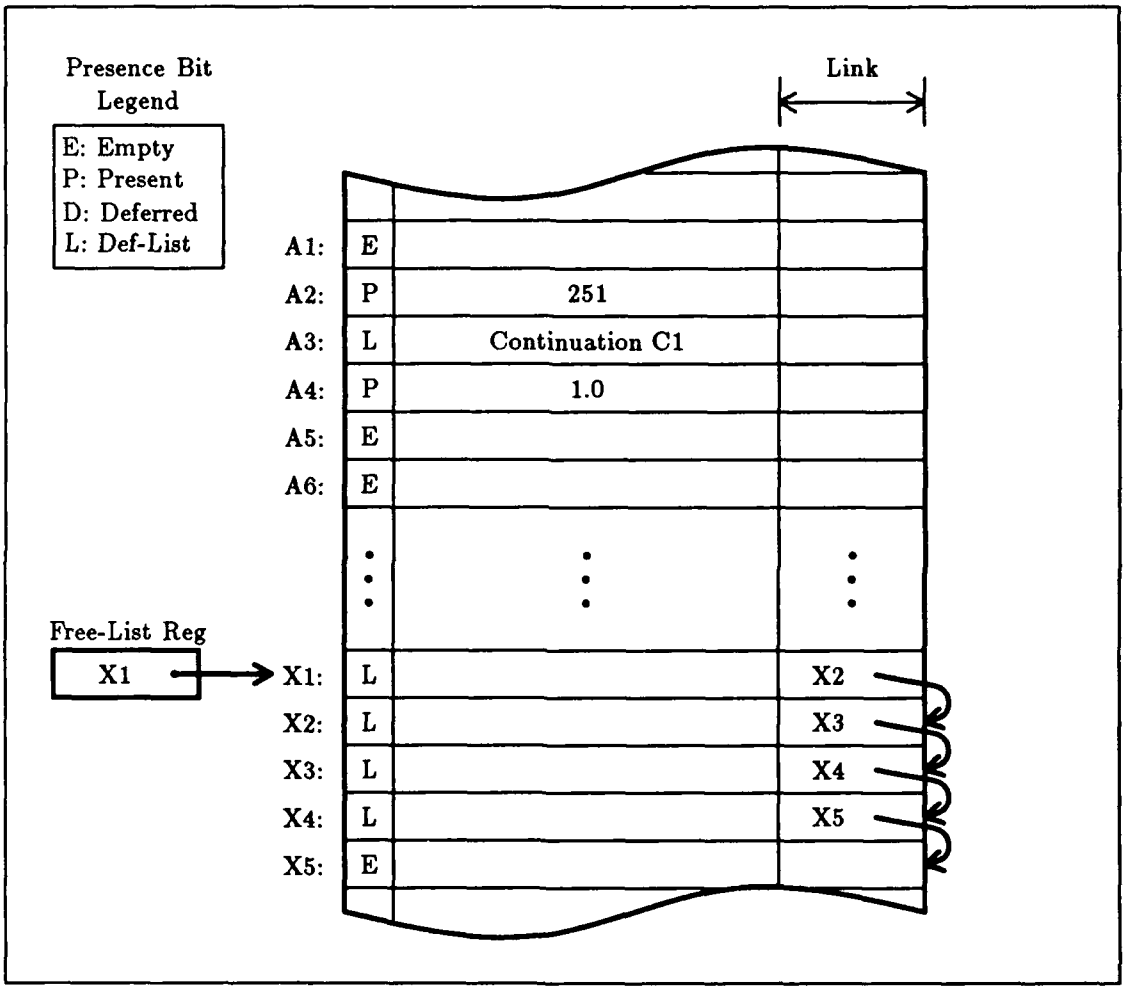


Figure 3.5: Single deferred request example in a memory with Link fields on every word. Each Link field can store a pointer local to the I-structure controller.

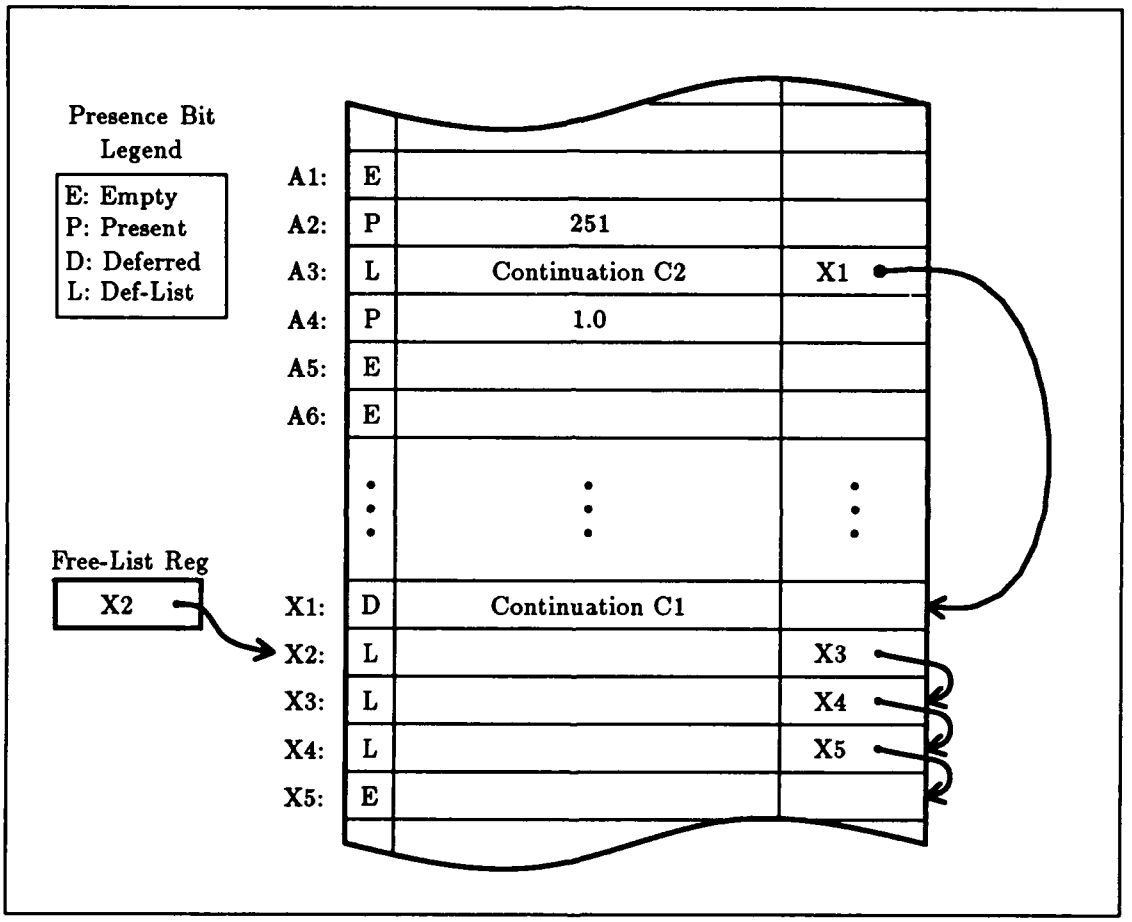


Figure 3.6: Local link deferred list example of Figure 3.5 after a second request is deferred.

The operations performed when a second request, R_1, with continuation C_1, arrives, change the memory state from that shown in Figure 3.5 to Figure 3.6. First a new cons cell word, X1, is allocated from the free list. The continuation, C_1, in location A3 is copied into location X1. On the same cycle that A3 is read, the continuation of the new request, C_2, and a pointer to the new cons cell word, X1, are written into the I-structure location, A3. The presence bits of X1 are set to *deferred* because the presence bits of location A3 were deferred. Location A3 is set to *deferred-list*, to indicate that the link field contains a valid pointer. The value read from the link field of the new cons cell word, X2, is stored into the free-list register, since it is now the first free cons cell word. The algorithm used to add a new request, R_i, to the deferred list is:

1. Exchange the new request's continuation, C_i, with the continuation, C_{i-1}, stored in the I-structure location. At the same, time exchange the free-list pointer with the value in the link field, call the value read L.

2. At the location pointed by the free-list pointer, read the link field into the free-list register and write the continuation C_{i-1}, and link pointer L. (This can be done with one exchange.)

Figure 3.7 shows the resulting memory state after a third request, with continuation C_3, is deferred. Two memory exchange cycles are required for each request added to the deferred list after the first, which only requires one exchange.

3.6.2 Satisfying Deferred Lists (Link Pointer Field)

The requests on the deferred list are satisfied when a value, V, is stored into the *deferred* location, A3, by I-STORE. If the location is in the *deferred* state, as in Figure 3.5 then the value V is exchanged with the deferred continuation, C_1, in the value field. The value, V, is sent to requesting process P_1 using C_1 and the location's state becomes *present*.

When a store is executed on a location in the *deferred-list* state, as in Figure 3.7, the value, V, from the I-STORE is exchanged with the deferred continuation, C_3, in the deferred location, A3 in the figure. V is sent to continuation C_3, which satisfies request R_3. The link pointer value, X2, is read along with the continuation, C_3, and used to address the

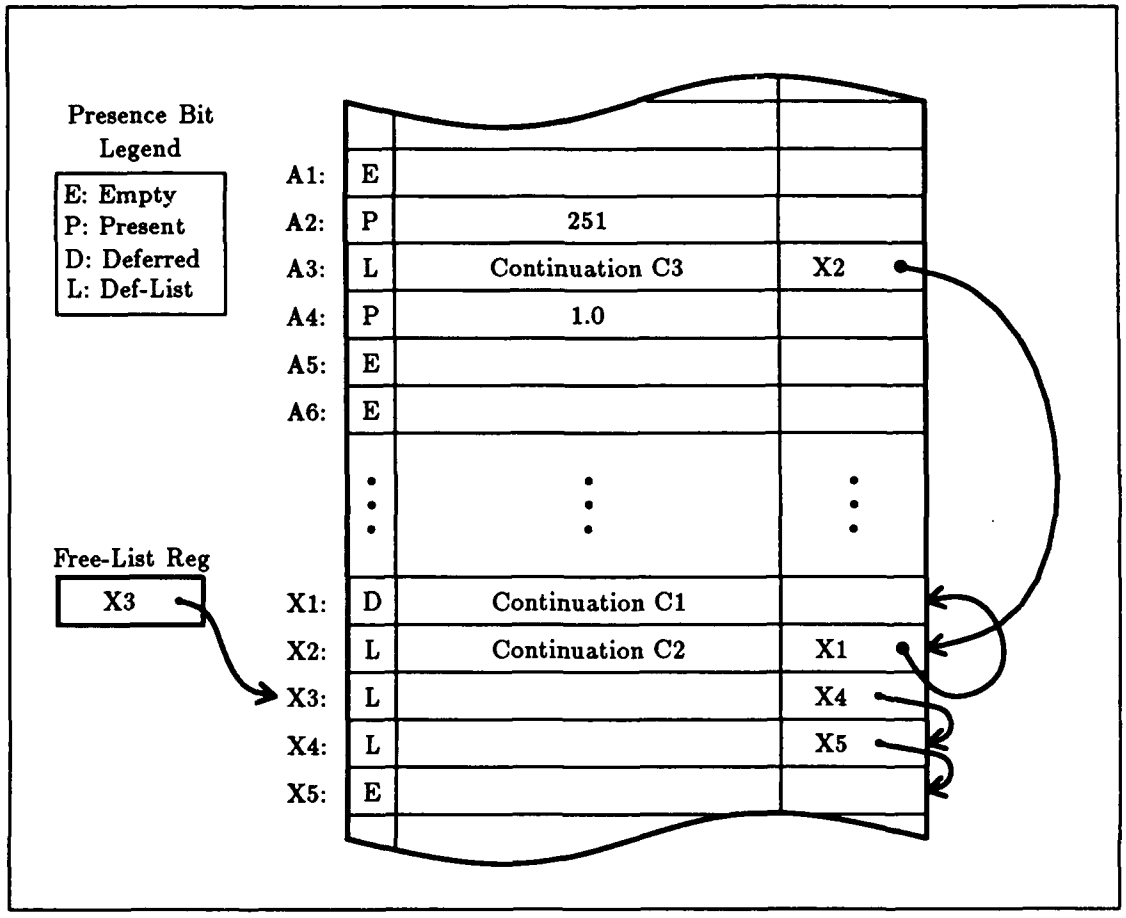


Figure 3.7: Local link example from Figure 3.5 after a third request is deferred.

next word on the deferred list; it is also saved in a temporary register for later use with the free-list. The next deferred continuation, C_2, and link pointer value, X1, are read from location X2, in one cycle. V is sent to the continuation, C_2. The process is repeated by following each link pointer until all of the elements on the list are satisfied; the end of the deferred list is indicated by a presence bit state of *deferred* instead of *deferred-list* on the cons cell word. In the example, the location accessed by the link pointer, X1, is *deferred.* The continuation read from location X1, C_1, is the last deferred continuation on the list, it also the continuation of the first request which to be deferred. The pointer in the free-list register, X3, is written into the link field of the last word of the list, X1. The temporary link register, pointing to the first cell of the deferred list, X2, is transferred into the free list register, thus pushing the deferred list onto the head of the free list. One memory cycle is needed to satisfy each element on the deferred list; one value is sent per cycle.

3.6.3 ID Code Simulation of Controller Operations for I-Structure

The ID code in Figure 3.8 illustrates the functions performed when I-structures are implemented using the local link list method for storing deferred lists. The code is very similar to the code for the local cons cell method presented in Section 3.5.3. The differences are in the handling of the *deferred* and *deferred-list* states. In the local cons cell method, every cons cell held two values and the deferred I-structure location contained only a pointer to the deferred list. Here, the deferred I-structure location also holds a continuation and the pointer to the deferred list except for the last cons cell, which contains only one continuation. This way, one send is issued on every cycle while satisfying a deferred list.

3.7 Comparing the Two Local Deferred List Methods

There are two basic differences between the two methods of storing local deferred lists. The first and most obvious is the way memory is used to create deferred lists. In both cases a deferred list of n deferred requests has $n-1$ list elements. The cons cell method uses two words per list element. The Link pointer method uses only one word per list element, so it uses less memory to store the bits, but the Link field is added to all memory words, even those not used to store lists. The break-even point between the two methods, in terms

```
% ID code simulating the functionality of I-Fetch and I-Store
% using the Local Link deferred list method.

type I_structure_location = present value | empty | deferred continuation
                          | deferred-list continuation address | error value;

def I_Fetch_op addr c =
    {case M[addr] of
          present v = {send c v}
     |    empty = {M[addr] := deferred c}
     |    deferred c' = {M[addr] := deferred-list c (alloc M[addr])}
     |    deferred-list c' a = {M[addr] := deferred-list c (alloc M[addr])}
     |    delayed trigger = { send trigger c;
                              M[addr] := deferred c}
     |    error v2 = {send Error_continuation v}};

def I_Store_op a v =
    {case M[addr] of
          present v' = { send Error_continuation v;
                         M[addr] := error v'}
     |    empty = { M[addr] := present v}
     |    deferred c' = { send c' v;
                          M[addr] := present v}
     |    deferred-list c a = { send c v;
                                Satisfy_def_list a v;
                                M[addr] := present v}
     |    delayed trigger = { send trigger v;
                              M[addr] := present v}
     |    error v' = {send Error_continuation v}};

def Satisfy_def_list a v =
    {case M[a] of
          deferred-list c a' = { send c v;
                                 Satisfy_def_list a' v}
     |    deferred c = {send c v}
                       M[a] := deferred-list c free_ptr}};

def alloc word =
    {case M[free_ptr] of
          empty = Free List Exhausted Error. Halting
     |    deferred-list foo a' = { free_ptr := a';
                                   M[a] := word}};
```

Figure 3.8: ID code: I-Structures with Local Link Pointer Deferred Lists

of memory bit usage, occurs when number of allocated deferred list cons cells is equal to half the number of words allocated for I-structures. In the Link field method, that means one third of memory is allocated as deferred list space. To get the same ratio of deferred list elements to I-structure words would require half of the memory in the local cons list method. In the Sigma-1 processor, which uses local cons cell deferred lists, one quarter of the memory space is allocated for deferred list storage, a ratio of 1 to 3. In that case the Link field method requires more memory than the cons method.

The second area of difference involves timing and control. Because both fields of a word are accessed in the same cycle in the Link field method, fewer memory accesses are required. The fact that the Link field is dedicated to storing the list pointers simplifies the I-structure controller's data path. Another consideration is the generation of output tokens. The cons cell method generates no outputs on the first cycle, since the I-structure location only contains a pointer, while on the last cycle two outputs are generated. The Link field method has simpler timing because it generates one output on every cycle. The longer words in the Link field method requires more memory to build but has simpler timing and control.

3.7.1 Advantages of Local Deferred Lists

There are several advantages of storing deferred lists locally on the I-structure controllers. First, it simplifies code, because every I-FETCH and I-STORE can be viewed as single operations; neither the processor or compiler need be concerned with deferring operations. It is a request-and-forget system; a processor sends a request and when the requested value becomes available it is returned to the processor. Second, network traffic is minimized. Each I-FETCH must generate one request token, which later results in one token to return the value. No extra tokens are generated by the local list method when a request is deferred. Third, the latency, measured from the arrival of the I-STORE, of satisfying a deferred I-FETCH request, is reduced to approximately one network transit time. This is discussed in more detail in Section 3.13. Fourth, Multiple-value Locks require local lists to store the list of deferred values.

3.7.2 Disadvantages of Local Deferred Lists

The local space used by the I-structure controller for deferred lists must be allocated beforehand. When the free deferred space is exhausted, the controller must stop processing tokens because it cannot handle another request that needs deferred space. This can result in a fatal system error, so there must be a mechanism for detecting the end of the free-list. Many schemes are possible that notice when the free list is getting low and inform a processor, but the free-list space could still be exhausted before the processor can allocate more. The problem of allocating deferred list space is further complicated because the space is distributed among many physically separate I-structure boards. Each controller can only use its local space. Sharing space across boards is difficult. Enough space must be allocated on each controller to handle the maximum space that it will need. The number of requests that are deferred depends the order of arrival of the fetch and store requests, which in turn depends on many run time factors, including how deferred lists are handled. The number of requests deferred by each controller, even given a known total number of requests, is affected by the allocation of memory across the controllers. Determining the correct deferred list space allocation is the most difficult problem of using local lists.

To support local lists, the I-structure controller must be able to perform sequences of operations. For example, satisfying a list of deferred I-FETCH requests requires walking down the list and sending a copy of the value to each deferred request. This requires more control hardware. Since all operations do not take the same amount of time, the variance of the controller's service time increases, which increases the waiting time of tokens in the input queue.

3.8 Distributed Deferred Lists

One way to eliminate the problem of allocating local deferred list space is not to store deferred lists locally. This could be done by disallowing any deferred reads, but that defeats the purpose of a synchronizing memory. One deferred read can be stored in the I-structure location. Allowing only one deferred read per location force the compiler to prevent multiple deferred reads, which may not be possible in all cases. As an alternative, the first deferred read could be stored and all others returned to the network to be retried later. This form

of busy-waiting does not require the processor's time, but does use network resources.

Is it possible to have deferred lists without keeping local lists on the I-structure controller? Instead of forming a local linked list of deferred continuations, the processes that made the requests are formed into a linked list. Each process stores one deferred continuation, the continuation of the next process on the deferred list. How the lists are formed and satisfied is explained in the following sections. Conceptually the idea is very similar to the local list methods, except instead of cons cells, the lists are constructed using requesting processes.

This method requires the cooperation of the requesting processes. Each requester stores one continuation on the deferred list. The I-structure controller stores one continuation on each deferred list in the I-structure location. Instead of storing the continuation of a new request locally, the I-structure controller sends the continuation it has stored to the process that issued the new request and stores the continuation of the new request. There are two methods for distinguishing the message containing a deferred continuation from the message containing the requested value these are referred to as the modified-value (Section 3.9) and modified-continuation (Section 3.10) methods. Each process that receives a deferred continuation stores it until it receives the requested value, then it sends the value to the stored continuation. In this way, the value gets propagated along the deferred list.

3.8.1 New Terminology

The message from the I-structure controller containing a deferred continuation that is sent to process P_i is referred to by the symbol D_i. The process stores the continuation contained in the message.

3.9 Distributed Lists using the Modified Value Method

There are now two distinctly different messages the I-structure controller can send to a process. As with the local methods, it returns the requested value, but it can also send a deferred continuation to be stored as part of the deferred list. This continuation must not be confused with the requested value. Since continuations are valid values to store in I-structure locations, the fact that the returned data is a continuation is not enough; another

way to distinguish deferred continuations from values is needed. One way is to modify the returned data in some way; that is the methods presented in this section. The type of the continuation is modified by the I-structure controller to be abnormal type; the requested value keeps its normal value type. The requesting process tests each value returned to see if it is normal or abnormal. abnormal values are stored as deferred continuations.

3.9.1 Creating Distributed Deferred Lists

The first request, R_1 from process P_1, is deferred by storing continuation C_1 in I-structure location A, exactly like the local methods. See Figure 3.9. The second request, R_2 from P_2, is deferred by storing its continuation, C_2, in A, which requires removing continuation C_1, since the location can only store one continuation. C_1 must be stored somewhere, so it is sent back to the new request's process, P_2, with its type set to abnormal to indicate that it is not the requested value. C_1 is returned to P_2 by sending it to continuation C_2. Process P_2 receives C_1 and since its type is abnormal, stores it for later use as a deferred continuation. See Figure 3.10. Subsequent fetch requests are deferred in the same manner. Request R_k, from P_k, is deferred by storing its continuation, C_k, in the requested I-structure location and sending the previously stored continuation, C_{k-1}, back to process P_k as an abnormal value, where it is stored. Figure 3.11 shows how the third request is deferred.

The result is a linked list of deferred processes. The head of the deferred list is stored in the *deferred* location on the I-structure controller. For a deferred list of length n, the I-structure location contains C_n, a pointer to process P_n. This is the same continuation that would be stored in the I-structure location in the Link method in Section 3.6. Process P_n stores continuation C_{n-1}, which points to process P_{n-1}. In general, any deferred requesting process, P_j for $j > 1$, stores the continuation C_{j-1} of process P_{j-1}. Process P_1, at the end of the list, does not store a deferred continuation.

3.9.2 Satisfying Distributed Deferred Lists

When a value, V, is stored into a *deferred* I-structure location, the I-structure controller exchanges the value, V, with the continuation, C_n, stored in the location and uses C_n to send V to process P_n. The same operation is performed independent of deferred list length.

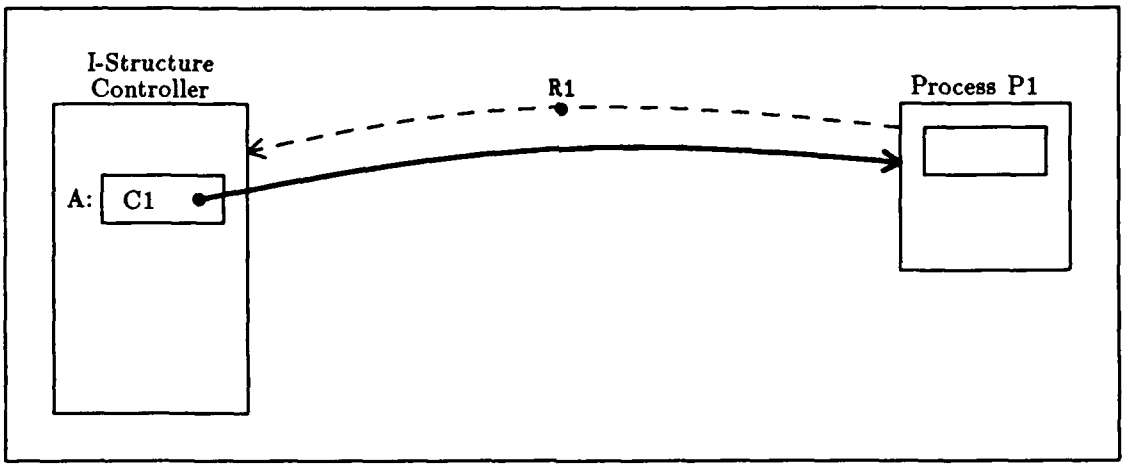


Figure 3.9: Example of distributed deferred list with one requesting process, P_1, sending fetch request R_1 to the I-structure controller. The controller defers the request by storing continuation C_1 which acts as a pointer to P_1. Solid arcs represent stored pointer; dashed arcs are messages.

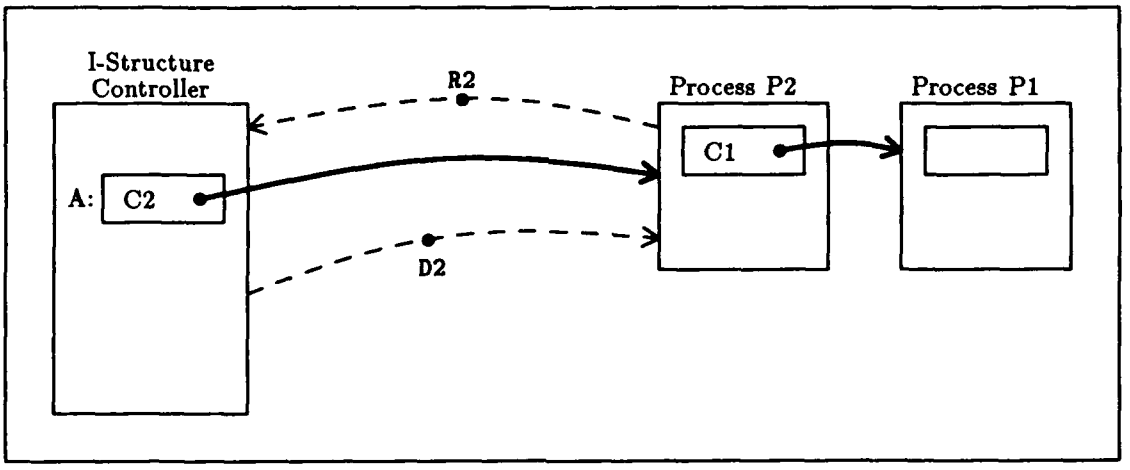


Figure 3.10: An extension of example in Figure 3.9. A second deferred request, R_2, from process P_2 is deferred by storing continuation C_2 and sending C_1 back to process P_2 in message D_2.

Each process, P_i, that receives the value, V, first tests its type. If it is a normal value type, then it checks if it has a stored deferred continuation; if so, it is continuation C_{i-1} which it uses to send a copy of V to P_{i-1}, the next process on the list. After performing both of these tests, then the process can use the value itself.

Each process on the deferred list, except the last, P_1, forwards a copy of the value, V, to the next process. The process at the end of the list, P_1 (the source of the first request deferred), does not forward the value since it never received a deferred continuation. Process P_1 acts as if its request was not deferred.

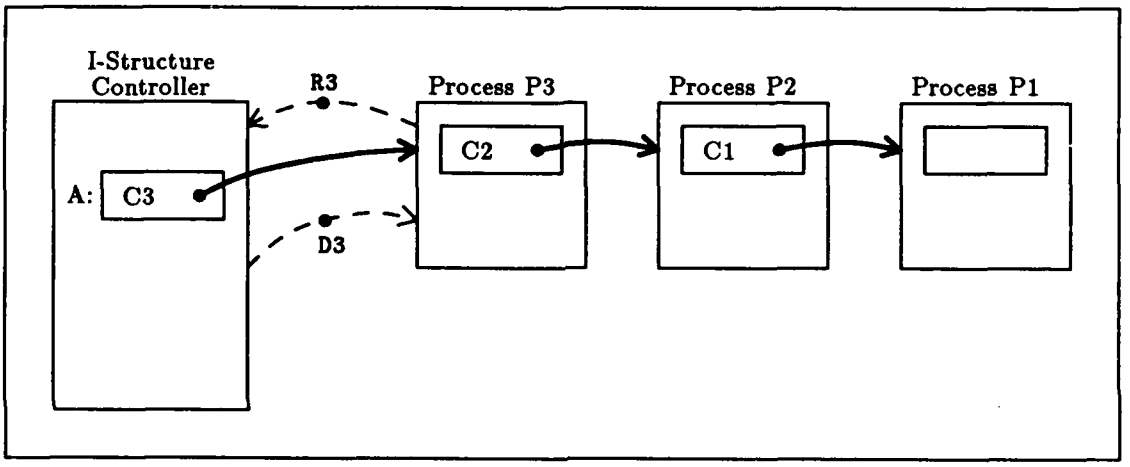


Figure 3.11: Extension of example in Figure 3.10 by a third deferred request R_3 from process P_3.

3.9.3 Id Code Simulation of Controller Operations for I-Structure

The code in Figure 3.12 simulates the operations performed by the I-structure controller when using the modified value method of constructing distributed deferred lists. The function *set_type* is used to modify the type of the deferred continuation that is sent back to the requesting process. Notice that there is no free-list management required. The free-list memory is handled by the processors.

```
% ID code simulating the operation of the I-structure controller when using
% Distributed deferred lists.

type I_structure_location = present value | empty | deferred continuation
                          | delayed continuation | error value;

def I_Fetch addr c =
    {case M[addr] of
          present v = {send c v}
     |    empty = {M[addr] := deferred c}
     |    deferred c' = { send c (set-type c' abnormal);
                         M[addr] := deferred c}
     |    delayed trigger = { send trigger trigger;
                             M[addr] := deferred c}
     |    error v' = {send Error_continuation c}};

def I_Store addr v =
    {case M[addr] of
          present v' = { send Error_continuation v;
                         M[addr] := error v'}
     |    empty = {M[addr] := present v}
     |    deferred c = { send c v;
                         M[addr] := present v}
     |    delayed trigger = { send trigger trigger;
                             M[addr] := present v}
     |    error v' = {send Error_continuation v}};
```

Figure 3.12: Id Code for Simulating I-structures using Modified value Distributed Deferred Lists.

3.9.4 Example Code for Process Issuing Request

In the requesting process, each I-FETCH must be accompanied by code that handles testing the returned value's type and managing the deferred list. The pseudo-assembly code in Figure 3.13 is an example of the kind of code needed. The processor has the option on each instruction to suspend execution of the current process, indicated by the *suspend* command. Due to the potentially long latency of I-FETCH, the processor should switch to a different task at each suspension. The function *make-request* takes as arguments a request type (e.g., I-FETCH) and an address. It returns a continuation that causes an I-structure controller to perform the requested operation on the given address. A continuation for the current process is created by the function *make-continuation*, the argument of which is the address of the instruction at which to resume execution when the I-structure controller returns a value to the process; any required context information is also included in the continuation. The *send* operation performs the same function as it does in the ID simulation code to the I-structure controller; a message is sent that starts remote execution, in this case on an I-structure controller. Whether these are built in hardware functions or subroutines is not important to this example. The value sent to the continuation that started execution of the code is represented by the variable, v.

```
                ⋮
        r = (make-request I-FETCH A);
        c = (make-continuation dest);
        def_flag := FALSE;
        send r c & suspend;    %Send request to I-structure controller
                ⋮
dest:   if (type-of v) == normal then dest2:
        stored_c := c;  % Store continuation of next process
        def_flag := TRUE & suspend;
dest2:  if def_flag == FALSE then dest3
        send stored_c v;  % Send value to continuation of next process
dest3:      %code to use fetched value
                ⋮
```

Figure 3.13: Code for Requesting Process using Distributed Deferred Lists

3.10 Distributed Lists using Modified Continuations

An alternative to modifying the value sent back to the requesting process is to modify the continuation to which the deferred continuation is sent. By sending the deferred continuation to a different continuation than the requested value the process does not need to test the value it receives. The requested value is sent to the original continuation and the deferred continuation is sent to a modified continuation. This requires the I-structure controller to understand the internal format of continuations; this was not the case for the local or modified value distributed methods. The continuation is modified to change the instruction at which the process will be restarted; no context information is changed. Modified continuations are represented as C_i' and C_i''. Given any of the continuations, the I-structure controller can convert it into any of the others.

The advantage of this method is that is does not require types or take away from the space of possible values. Also, the code executed by the process is simpler because if does not need to test the received values. Example code is given in Figure 3.14. For illustrative purposes, the continuation is modified by subtracting from the address of the instruction pointer.

The processes are linked into the same lists and continuations of the same processes as in the modified value method of distributed lists; only the continuations used to send messages are different. A deferred continuation is stored by sending it to the modified continuation, C_i', to be stored by process, P_i. The requested value is sent to C_i'' if process P_i was sent a continuation, otherwise it is sent to the unmodified continuation, C_i. A process either receives one value sent to continuation C_i, or two values, one sent to C_i' and the other to C_i''.

3.10.1 Creating Distributed Deferred Lists

The first request, R_1 from process P_1, is deferred by storing continuation C_1 in the I-structure location; the continuation is not modified. See Figure 3.9. The second request, R_2 from P_2, is deferred by storing continuation, C_2'', in A, and removing continuation C_1. C_1 is sent to C_2'. Process P_2 receives C_1 and the code at the C_2' entry point stores it, in $stored_c$, for later use. See Figure 3.14. The continuation of the next request deferred,

C_3, is deferred by sending the continuation stored in the I-structure location, C_2'', to C_3' and storing C_3'' in the I-structure location. Process P_3 stores continuation C_2''; it does not matter that is it a modified continuation. Subsequent fetch requests are deferred in the same manner. Request R_i, from P_i, is deferred by storing its modified continuation, C_i'', in the requested I-structure location and sending the previously stored continuation, C_{i-1}'', to C_i' were it is stored by process P_i.

All of the stored continuations are of the modified C'' type except C_1 which is stored by P_2; P_1 is not used to store a deferred continuation so it is sent the requested value directly. The head of the deferred list is stored in the *deferred* location on the I-structure controller. For a deferred list of length n where $n > 1$, the I-structure location contains C_n'', a pointer to the deferred value entry point in process P_n. In general, any deferred requesting process, P_i for $i > 2$, stores the continuation C_{i-1}'' of process P_{i-1}. Process P_2 stores C_1. Process P_1, at the end of the list, does not store a deferred continuation.

3.10.2 Satisfying Distributed Deferred Lists

Distributed deferred lists are satisfied the same way independent of which method is used. The I-structure controller controller sends the value to the continuation stored in the I-structure location. Each process forwards a copy of the value to its stored continuation. Whichever entry point is started by the continuation correctly handles forwarding the value. The unmodified continuation, C_1, used to send the value to the last process on the list, P_1, ends the list by virtue of the fact that the unmodified continuation entry point code does not forward the value.

3.10.3 Example Code for Process Issuing Request

An example of code, in pseudo assembly style, executed by the processes for each entry point of each modified continuation is given in Figure 3.14. The unmodified continuation entry point, Dest, is simply the code to use the fetched value. The first modified entry point, Dest1 entered by C_i', simply stores the deferred continuation in the variable *stored_c*. The second modified entry point, Dest2 entered by C_i'', is entered with the requested value if a deferred continuation is stored in *stored_c*. It forwards a copy of the received value, v to

the stored deferred continuation, *stored_c*, and then jumps to the unmodified entry point, Dest.

This code is much simpler than the code given in Figure 3.13 for the modified value distributed list method.

```
                ⋮
        r = (make-request I-FETCH A);
        c = (make-continuation Dest);
        send r c & suspend;    % Send request to I-structure controller
                ⋮
Dest1:  stored_c = v & suspend
Dest2:  send stored_c v;
Dest:   % Code to use fetched value
                ⋮
```

Figure 3.14: Processor Code for Modified Continuation Distributed Deferred Lists

3.11 Distributed Deferred Lists for Locks

The Lock synchronization mechanism requires the ability to remove (pop) one deferred request from a deferred list. This is needed because the lock is only given to one of the deferred requests, the others remain deferred. For local lists this is all done atomically on the I-structure controller, but when done distributedly it is more complicated.

The I-structure controller defers TAKE requests in exactly the same way that I-FETCH requests are deferred with distributed lists. When a value is PUT back to a *deferred* Lock location, the value is sent to the continuation stored in the location; the location's state becomes *empty*, even if there is a list of deferred requests. When the first process on the deferred list receives the lock's value it sends its stored deferred continuation, if it has one, back to the lock location in a TAKE request on the lock location; the stored continuation, C_i, is the requests continuation. This is equivalent to the original process P_i re-issuing the TAKE request. When that TAKE request arrives at the I-structure controller it is deferred again. If the lock location is *empty* then the continuation is stored in the I-structure location. One value has been removed from the deferred list.

The description just given is not the only possible scenario. Removing a request from a distributed deferred list is not an atomic operation; others can modify the lock location during the process. While one continuation is being removed from the list, and the lock location is *empty*, another TAKE request can be deferred. In the I-structure distributed list method, the stored continuation is sent to the process of the arriving TAKE request, because it is guaranteed that every arriving request has space to store one deferred continuation. Here, an arriving request might be a deferred list; the remainder of a deferred list that has had its first request popped off by a PUT request. Then the process of the arriving request already has a deferred continuation stored; the pointer to the rest of the deferred list. Why not send the arriving request to the process of the continuation stored in the Lock location? That continuation could itself be the beginning of a list of deferred continuations. If a process which already has a stored deferred continuation is sent a second deferred continuation it won't have any place to store it. The last process of each of these list does not have a stored continuation, so it has free space in which to store a continuation. The problem is that the continuation of the last process is not known by the I-structure controller, only by the second to last process on the list. The last process can be reached from the beginning by following all of the continuation pointers in the list.

By modifying the action taken by the processes when they receive a second deferred continuation, one deferred lists can be appended to another. A process which receives a second deferred continuation, C_a, when it already has a stored deferred continuation, C_s, sends C_a to C_s as a deferred continuation to be saved by process P_s; it passes the buck (continuation) to the next process on the list. Each process on the list does the same thing until the last. The last process on the list does not have a stored deferred continuation, so it saves the continuation that has been passed down the list. This appends the list starting with process P_a to the other list. Which list is appended to the other depends on whether the arriving continuation is sent to the stored or vis versa.

There is another option; the two lists can be merged. Merging is accomplished by having each process exchange an arriving deferred continuation, C_a, for its stored deferred continuation, C_s. It then sends C_s to C_a. Merging has the feature that the number of messages is two times the order of the shorter list. This is good if one list is much shorter than the other, but worse than appending if both are approximately the same length. Merging also requires the process to perform an exchange, where as appending only requires

are read.

3.11.1 Network Triangle Inequality Problem

Along with the requirement of point-to-point FIFO ordering in the network, distributed TAKE lists have further requirements. The problem occurs because of the triangle inequality problem discussed in [13], which states that the network path from node A to node B may be slower than going from node A to node C and then to node B. This means that there is no FIFO ordering between messages issued by node A and those received by node B, unless they are all sent by the same path.

The triangle inequality problem enters into distributed list because of the danger of a value getting ahead of the process of propagating an append or merge along the deferred list. A process does not expect to receive any deferred continuations after the requested value. This is necessary to allow the process to terminate. The append operation travels between processes along the deferred list, but the lock's value returns to the lock location between each process. Since these are different network paths no guarantee about ordering can be made. Some other means is required to guarantee all continuations arrive before the value.

The solution is for the processes to add extra synchronization where it is needed. Each process, P_i, which sends a continuation, C_k, to another process, P_j, waits for an acknowledgment that the continuation, C_k, has been received by the next process. Until it receives the acknowledgment it will not send its stored continuation, C_j, back to the lock's location. This prevents the value from arriving at the next process, P_j, before the continuation C_k, sent by process P_i because the value cannot be sent to process P_j until its continuation, C_j, is returned to the lock location on the I-structure controller. The network should provide an acknowledgment mechanism.

In a network without acknowledgments, an alternative to appending or merging the deferred lists is to tear apart one of the lists and have each of the processes on the list send a new TAKE request to the lock location. This is a form of limited busy-waiting, but it avoids the triangle inequality problem.

3.12 Summary of Distributed List Method

3.12.1 Similarities with Previous Systems

The distributed method of storing deferred lists presented here uses basically the same mechanism used to handle multiple demands on a lenient cons cell in the demand driven system presented in [10]. There the idea is called forward chaining. The first time a location is demanded a forwarding pointer, to a local location, is left in the location of the demanded value. The demand starts the execution of code to calculate the value demanded. When the value is evaluated it overwrites the forwarding pointer, which causes a copy to be forwarded via the forwarding pointer to the local location. In distributed lists the forwarding pointers are continuations and fetch requests are demands. Delayed triggers are used to implement this kind of lazy evaluation. The second demand of a location that has already been demanded replaces the forwarding pointer in the location with a forwarding pointer to the new demander's cell. The old forwarding pointer is stored in the new demander's cell. This corresponds to pushing a continuation onto a distributed deferred list.

3.12.2 Advantages of Distributed Deferred Lists

Using distributed deferred lists has the following advantages. First, the problem of allocating enough deferred space on each I-structure controller is eliminated because there is no local deferred list storage. The deferred list space allocation is combined with the allocation of the processors local memory so only one resource must be managed. A process cannot make a request unless it has the space to store a deferred continuation, so it is impossible to overflow the deferred list space.

Second, all of the I-structure controller's memory can be used for I-structures. The amount of deferred space needed is a function of the current number of requests, not the total size of I-structure memory.

Third, the I-structure controller's hardware is simpler because it does not need to manage local lists, that requires the ability to perform sequences of operations. Each input message results in at most one memory cycle and one output message.

Fourth, the compiler is given complete control over allocating deferred space. This allows room for optimization. If there are cases where it can prove that a request will not

be deferred, then it does not need to allocate deferred space. The same mechanisms that are used to control parallelism can be used to control the number of deferred reads.

3.12.3 Disadvantages of Distributed Deferred Lists

First, there is more processor and network overhead when using distributed deferred lists. Some of the I-structure controller's work has been shifted to the processors. All of the code executed on the processor to handle deferred lists is overhead. Each request deferred generates an extra message to send the old deferred continuation back to be stored in the requesting process. In fact, all of the messages to send deferred continuations are overhead of the distributed list method. It is still better than a system that forces the processor to retry its requests. At least two message are required for a retry; one to inform the processor to re-issue the request and the new request message. If the retry request also fails that causes an additional two messages. Only one extra message is required to add a process to a deferred list.

Second, the latency to satisfy a deferred list is longer because the value must travel between processors on every hop in the deferred list. In practice this is not always true because some adjacent processes on the list may reside on the same processor, but with a large number of processors this is a small percentage.

Third, the deferred lists are stored in the processor memory, which is smaller, faster and more costly than the I-structure controller's memory, that makes it a more limited resource.

Fourth, since deferred space must be allocated for every request but not all requests are deferred, there is always more space allocated than used. Also, the first deferred request's continuation is stored in the I-structure location, but deferred space is still allocated for it locally on the requesting process. This is required because which request will arrive first is not known. Since the deferred space can be reclaimed as soon as the requested value is returned, this extra deferred space is a dynamic quantity. With local lists, the deferred space must be statically allocated beforehand and it must be larger than the maximum amount actually used.

3.13 Latency Trade Offs Between Local and Distributed Lists

The time to satisfy a given deferred request, from the time the value is stored, is the time to process all of the elements on the list before the given deferred request plus the time for the value to travel to the deferred requester. For local deferred lists, the time to process each element on the list is the memory cycle time of the I-structure controller, denoted t_{is}. The travel time is one network transit time, denoted t_{net}. So, for the n^{th} element on a local deferred list the latency is:

$$n\ t_{is} + t_{net}$$

For deferred lists stored in frames, the time to process an element on the list is one network transit time, plus one processing element pipeline trip, to execute the I-DEFER instruction, of t_{pe}. The travel time is one network transit time, unless it is on the same processor, in which case it is the time spent waiting on the processor's token queue. This analysis assumes the worst case of different PEs every time, so a network transit time is included. So, the n^{th} element of a frame deferred list has a latency of:

$$t_{is} + t_{net} + (n-1)(t_{net} + t_{pe})$$

The extra latency for the activation frame method is then:

$$(n-1)(t_{pe} + t_{net} - t_{is})$$

The local deferred list method has a lower latency provided that $t_{is} < t_{net} + t_{pe}$, which is true in the current design, t_{net} dominates.

Chapter 4

Current Design

The final product of the work discussed so far is the hardware design specification for the I-structure controller, included in appendix A. This chapter details the use of the I-structure hardware for implementing the synchronization methods from Chapter 2 as part of the Monsoon dataflow computer system. Those interested in hardware details should see the appendix. The I-structure controller is being built by a group at Motorola Microcomputer Division in Tempe, Arizona, as the memory for the Monsoon computer system.

Practical realities had important effects on the design. A minimal hardware design is important since it decreases design time and increases the probability of success. The project schedule of the Monsoon processor required a quick memory controller design. A complete system cannot be tested without the I-structure memory. The need for simplicity improved the design by requiring an understanding of the essential support needed for the synchronization methods. Since the amount of available board area is limited in a real implementation, the smaller the controller, the more area available for memory, the reason for the controller.

4.1 Design Decisions

Two problems lead to the choice of using distributed deferred lists. The major problem, that of overflowing locally available deferred list space was not seen to have a clear solution. Hardware solutions, such as sharing deferred space between boards, are complicated and remove any flexibility from the software. Allocating sufficient storage statically is a hard

problem because the number of reads that are deferred is hard to model and the effect of overflowing is fatal. The distributed list methods removes the problem of allocating local deferred list space by eliminating the local lists. A secondary problem with any local list method is that satisfying a deferred list requires multiple cycles to read the deferred list and generate the output tokens. This complicates control and results in variable execution times for all of the operations. The increased variance of the execution times can cause the input FIFO to fill up, which could block the network. This is solved on the processors by dumping the input FIFO into the token queues, which are much larger than the FIFO, when it becomes full. The distributed list method allows an I-structure design which always takes only one cycle and generates only one token per request.

It is hoped that in the future the number of reads deferred can be reduced with software solutions. There is no benefit gained from a read which is deferred for a long time; in fact, it ties up resources. The distributed list method leaves all of the control with the software, which allows room for optimization. This method of deferred list storage allows for further reduced complexity of the I-structure controller.

The Monsoon processor was designed so that a separate I-structure controller was not required. On the Monsoon prototype, special instructions are used to emulate I-structures. The method used for storing deferred lists is a specialization of the distributed deferred list method presented in Section 3.8. More accurately, the distributed list method is a generalization of the Monsoon method, because the latter was developed first. The storage of deferred lists and the emulation of I-structures on Monsoon is covered in Section 4.4.

4.2 A Stripped Down Monsoon Processor

The design of the I-structure controller is very closely related to the Monsoon processor. This was done to maintain compatibility and to use the good ideas in Monsoon. The I-structure controller can be viewed functionally as a stripped down processor. Of Monsoon's eight pipeline stages, only three are needed: Presence Bits, Frame Store, and Form Token. The functionality of the Form Token sections is further reduced to generate only one token per cycle. All of the arithmetic logic, instruction memory, and token queues are removed.

To accommodate the longer cycle time of dynamic memory, the three Monsoon pipeline

stages have been collapsed into one with a longer cycle time. The cycle time of the new stage is less than the three Monsoon stages, but more than one Monsoon stage, so it has a lower throughput rate. The throughput rate is matched to the network's token delivery rate; tokens are processed by the I-structure controller at the maximum rate they can be delivered. The network used in the Monsoon Computer is a packet routing network composed of packet routing chips, called PaRC (Packet Routing Chip) [9]. For more details see Section A.3.3 in the appendix.

Because of the history that the I-structure controller was developed after the Monsoon processor many of the names from Monsoon have been preserved. In a tag the Instruction Pointer (IP), which on Monsoon is the address on an instruction, is used to directly encode the I-structure controller opcode. A local memory address is referred to as a Frame Pointer (FP). Network node addresses are called Processing Element (PE) numbers whether the node is an I-structure controller or a processor.

4.3 Why Build an I-Structure Controller?

The question has been asked, "Why build an I-structure controller at all when the same functions can be emulated by a processor?" The first reason is to provide more memory. Dynamic memory has a lower cost and higher density per bit than the static memory used in the processor. The static memory on the processor must be fast, and thus more expensive, due to the shorter cycle time of the processor pipeline. Dynamic memory requires periodic refreshing, which is difficult to integrate into a pipelined processor. The I-structure controller is specialized to handle the longer cycle time and more complex control timing of dynamic memory.

Second, the I-structure controller is simpler, requiring less hardware and leaving more space available for memory chips. This results in an I-structure memory sixteen times larger than the processor memory: 4 Megawords as compared to 256 Kilowords. With the next generation of memory chips, the plan is to upgrade I-structure and processor memories to 16 Megawords and 1 Megawords respectively; at nine bytes per word these are not small memories.

Third, the latency of the I-structure controller is lower; on Monsoon all tokens must

traverse all eight pipeline stages, even if they are only affected by three. The latency of the I-structure controller is determined by the cycle time of the dynamic memory, which is less than three Monsoon pipeline ticks.

Fourth, the instruction overhead required to emulate I-structures is shifted from a processor to an I-structure controller, allowing the processor to be better utilized for computation.

4.4 Distributed Deferred Lists Storage on Monsoon

The method used to store deferred lists on Monsoon is a specialization of the distributed deferred list method presented in Section 3.10. This method was designed into the Monsoon prototype to allow it to emulate I-structures. There are two parts to the distributed list method, one performed by the processors making requests and one performed by the I-structure controller, or processor emulating an I-structure controller, responding to the requests. The I-DEFER instruction presented in Section 4.4.1 handles all of the work the process does related to deferred lists. The operations performed on the memory location affected by the requests are handled by the I-structure controller. This is detailed in Section 4.7.

4.4.1 The I-DEFER Instruction

Implementing the distributed list methods on Monsoon requires the compiler to add an extra instruction for each I-FETCH instruction it compiles into a program graph. This extra instruction, called I-DEFER, handles storing the deferred list. The I-DEFER instruction appears immediately before the instruction to which the fetched data is returned. See Figure 4.1. The deferred continuation sent in the I-FETCH request (token T1 in Figure 4.1) points to the destination instruction (+). The I-DEFER is a dyadic (two input) instruction. It is executed only when it is used to store part of a deferred list, as in Figure 4.2. Each I-DEFER plays a roll similar to the CDR of the cons cell in the other two linked list methods; it stores the continuation of next process on the deferred list in its one activation frame slot, a word of local processor memory dedicated for use by the instruction while it is waiting to execute.

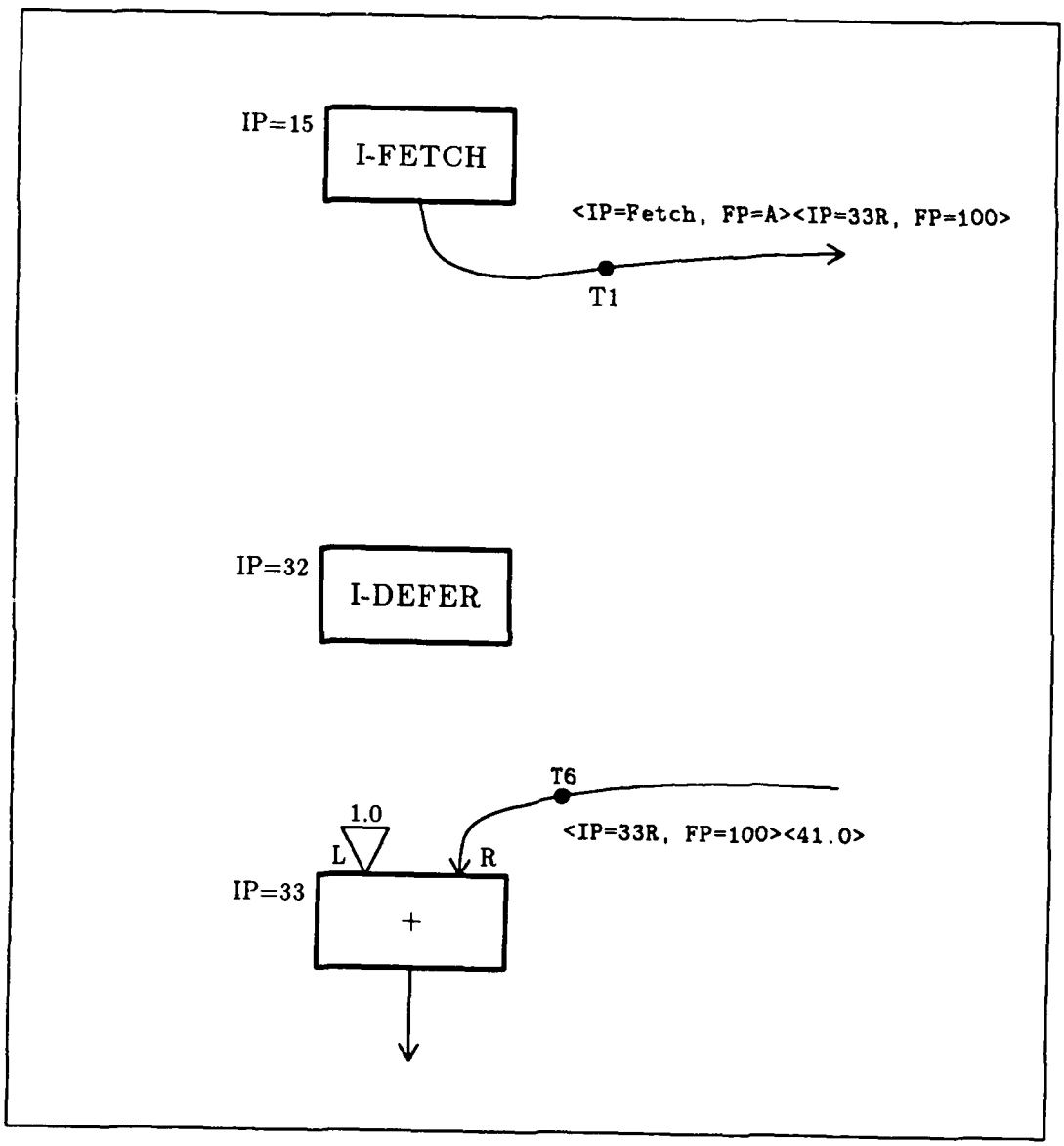


Figure 4.1: Example program graph with an I-DEFER instruction that is not used to store a deferred continuation.

The I-DEFER's two inputs are the deferred continuation (T2) and requested value (T3) tokens generated by the I-structure controller. These two tokens are always generated as a pair, or not generated at all.

The modified continuations (C' and C'') are generated from the original continuation ($< IP,\ FP >$) by subtracting one from the IP. Actually, to simplify the hardware this is implemented by flipping the least significant bit of the IP and requiring that the IP of the destination instruction be odd; the net result is then the same as subtracting one. The two modified continuations are differentiated by their port bits (C' is left and C'' is right).

The I-DEFER instruction does not execute until it receives both of its input tokens (this is the normal dyadic matching mode). When the I-DEFER is executed it sends one copy of the value it received (in T3) to the next instruction (the +), which is the destination of the deferred request. Another copy of the value is sent (in T4) to the stored continuation which was received as the other input to the I-DEFER (T2). That continuation is either the continuation of the next I-DEFER on the list, or the continuation of the destination instruction in the last deferred process on the list. The I-DEFER instruction never modifies a continuation. Only the I-structure controller does that.

4.4.2 Reuse of Frame Slots

The frame slot allocated to the I-DEFER instruction can be re-used as soon as its associated destination instruction receives its value. The value can be received from only one of two places: the I-DEFER instruction, or the I-structure controller. By the rules used to form dataflow program graphs, a frame slot is only required by an instruction which is waiting to execute. Two instructions can share the same frame slot provided they can never be waiting at the same time. If the value came from the I-DEFER then it has executed because it only generates outputs when it executes. On the other hand, if the value came from the I-structure controller then the I-DEFER instruction will never execute since it can only be used to add its associated request to the deferred list; that request has been satisfied. An instruction that never receives any input tokens does not need its frame slot. The I-structure controller either sends both or neither of the input tokens to an I-DEFER.

There are some cases in which the I-DEFER can reuse the frame slot of another instruction. If the I-FETCH instruction is dyadic, and neither input is a constant, then its frame

slot can be used by the I-DEFER instruction. No tokens are sent by the I-structure controller before it receives the request token (T1), which is sent only after the I-FETCH instruction executes. Thus, the I-DEFER instruction cannot receive any input tokens until the I-FETCH has completed, so it is safe for it to use the same frame slot as the I-FETCH.

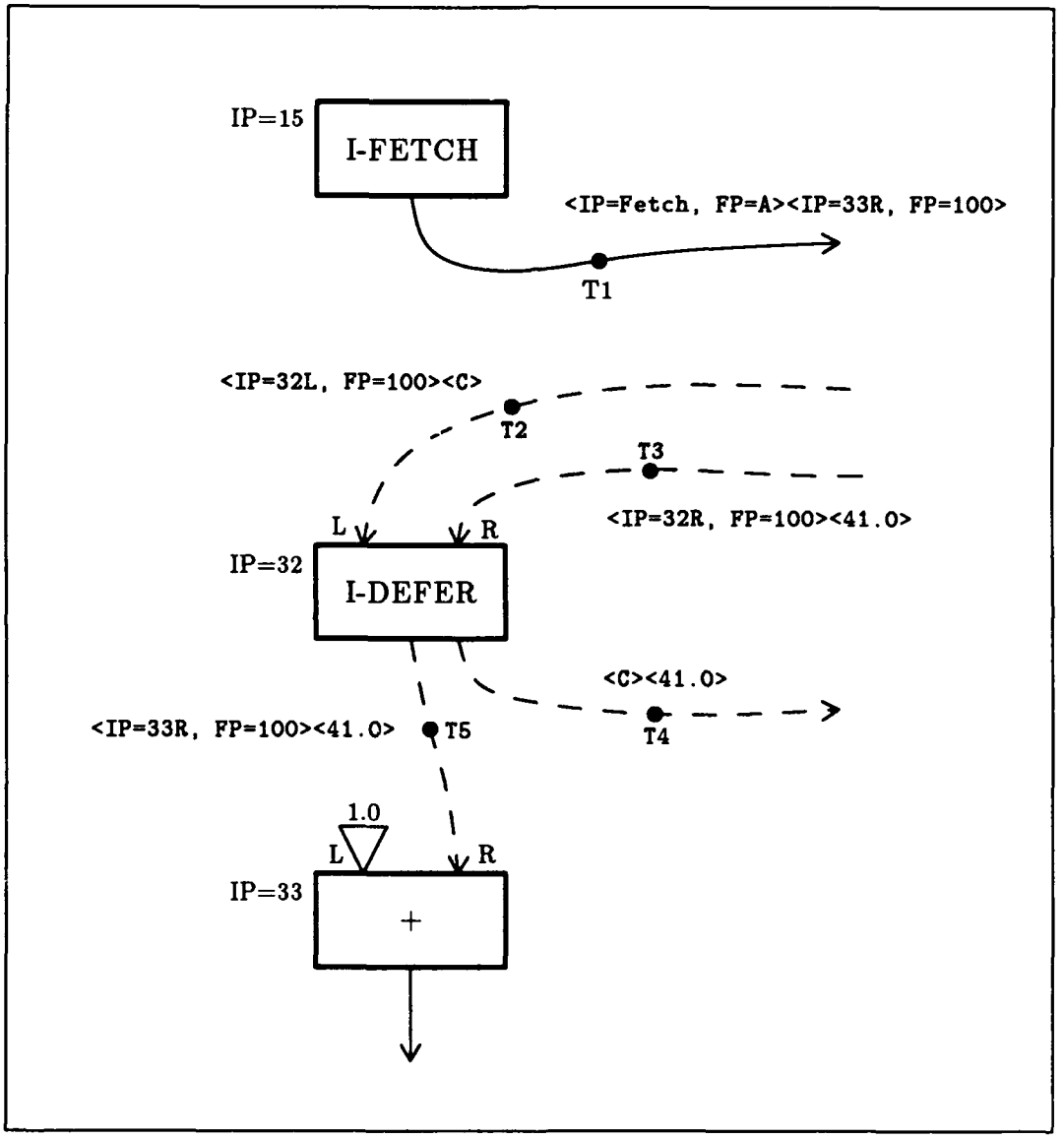


Figure 4.2: Example program graph with an I-DEFER instruction used to store a deferred continuation.

4.5 Instruction Overhead

Storing deferred lists in the activation frames of the processor requires executing extra instructions on the processor. For an n element deferred list, $n-1$ I-DEFER instructions are executed. Each I-DEFER is a dyadic instruction, which require two processor cycles to execute, one for each input token. These instructions are only executed if the request is deferred.

The latency and instruction overhead are both dependent on the number of fetches that are deferred. Fetching *present* data does not defer and does not incur the overhead of executing any I-DEFER instructions. How many fetches are deferred is highly dependent on the actual order in which the program is executed. With asynchronous processors, that could change with each execution of the program.

4.6 Monsoon Distributed Deferred List Examples

Some examples should help to clarify how distributed deferred list are stored on Monsoon. The first example sets up one deferred request, shown in Figure 4.3, and then shows the results of satisfying that request. The second example builds on the first by adding a second deferred request to the list created in the first example. See Figure 4.4. The process of satisfying this deferred list is then described.

4.6.1 First Fetch Example

The I-structure location with address X starts in the *empty* state. An I-FETCH request, R_1 is sent by process P_1 to the I-structure controller. The I-structure controller defers the request, because the location is *empty*, by storing its continuation (IP1.FP1) in location X and setting the presence bits to *deferred*. The diagram in Figure 4.3 illustrates this situation. The I-structure location contains the continuation (IP1.FP1) which points to the destination instruction, at IP1 in activation frame FP1, which is waiting for the value of location X. When a value is written to location X it is also sent to the stored continuation. In the figure the value follows the path of the solid arc.

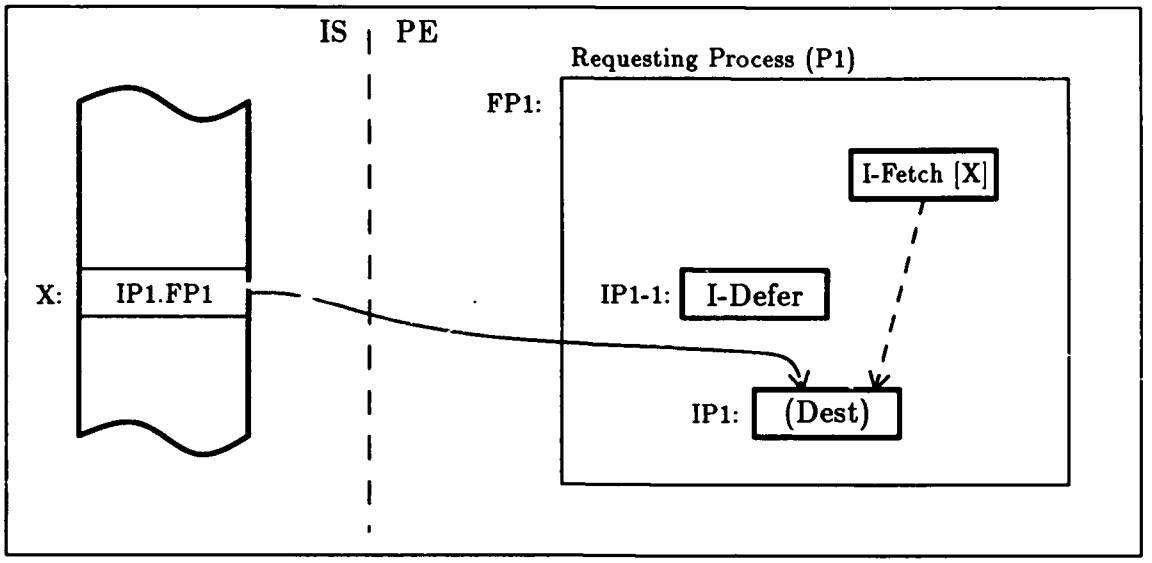


Figure 4.3: Example of single deferred request on location X. The continuation (IP1.FP1) is the continuation of the destination instruction in process P_1.

4.6.2 Second Fetch Example

A second I-FETCH request, sent by process P_2, for location X arrives at the I-structure controller. The second request contains continuation (IP2.FP2). Location X is *deferred* and contains the continuation (IP1.FP1) of the first process. Because the location is *deferred*, the I-structure controller modifies the continuation (IP2.FP2) by flipping the LSB of IP2 to give continuation (IP2-1.FP2) which is exchanged with the continuation (IP1.FP1) in location X; the presence bits remain *deferred*. The I-structure controller generates the output token $< IP2-1.FP2 >< IP1.FP1 >$ with port set to left. The I-DEFER instruction in process P_2, at instruction address IP2-1, receives the token and since it must have both inputs to execute it saves the continuation (IP1.FP1). The resulting situation is illustrated in Figure 4.4. When the other input, the requested value, arrives the I-DEFER instruction executes and sends the value to its destination instruction (IP2.FP2) and to the stored continuation (IP1.FP1). The value is sent to the I-DEFER instruction in process P_2 by the I-structure controller when the value is stored into location X. The I-structure controller stores the value and sends a copy to the deferred continuation stored in the location.

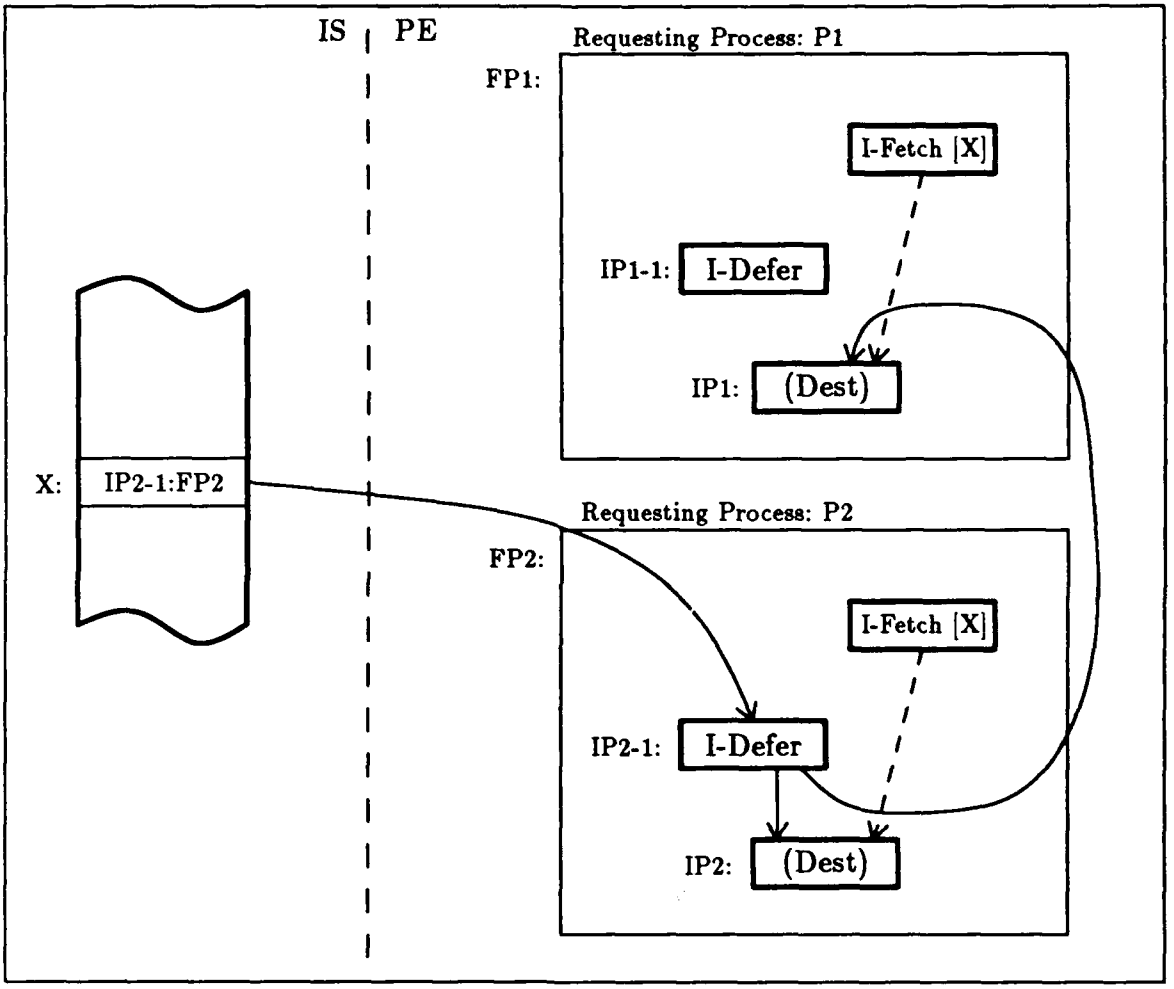


Figure 4.4: Example of distributed deferred list with two deferred requests on location X. The continuation (IP1.FP1) is an unmodified continuation. (IP2-1.FP2) is a modified continuation.

4.7 I-Structure Controller Operation

The I-structure controller receives request tokens over the network. Each token has a tag-part and a value-part. The value-part may be either a program value (for write-like operations) or a continuation (for read-like operations). The tag-part contains the address of the word to be accessed (the frame pointer or FP) and an opcode for the operation to performed (the Pmap which stands for presence bit map). Details of the layout of fields are included in figures A.1 and A.2 in Section A.3.1 on page 111.

The Pmap (opcode number) is used to determine what type of synchronization to use (e.g., I-structures). For each method there is a read-like and write-like operation (e.g., I-FETCH and I-STORE for I-structures), which share an opcode and are distinguished by the port bit. The port can be either left (port=0) for read-like operations or right (port=1) for write-like operations. This assignment is a compiler convention and not dictated in any way by the hardware. There are two classes of operations: one used for the synchronization methods and the other for system level support functions. All of the operations supported by the I-structure controller are listed in Table 4.1 and Table 4.2. Each line in the tables represents one opcode.

Normal Opcodes		
Pmap	Port	
Type of Synchronization	Left	Right
I-Structures	I-FETCH	I-STORE
Locks	TAKE	PUT
Examine Lock	EXAMINE-LOCK	
Delayed Triggers		STORE-DELAY
Imperatives	READ	WRITE

Table 4.1: Normal Opcodes

4.7.1 ID for Emulating I-Structure Opcode

The following ID code simulates the behavior of the I-structure controller in response to I-FETCH and I-STORE requests. Distributed lists which modify the continuation are used to store the deferred lists.

```
% ID code demonstrating the functions performed by the I-structure controller
% opcode which implements I-Fetch and I-Store.
% Distributed deferred lists are used.

type I_structure_location = present value | empty | deferred continuation
                          | delayed continuation | error value;

def I_Fetch_op addr c =
    {case M[addr] of
          present v = {send c v}
     |    empty = {M[addr] := deferred c}
     |    deferred c' = { send (set-port (dec-ip c) left) c';
                         M[addr] := deferred (dec-ip c)}
     |    delayed trigger = { send trigger c;
                              M[addr] := deferred c}
     |    error v2 = {send (set-fp Error_continuation addr) c}};

def I_Store_op addr v =
    {case M[addr] of
          present v' = { send (set-fp Error_continuation addr) v;
                         M[addr] := error v'}
     |    empty = {M[addr] := present v}
     |    deferred c' = { send (set-port c' right) v;
                         M[addr] := present v}
     |    delayed trigger = { send trigger v;
                              M[addr] := present v}
     |    error v' = {send (set-fp Error_continuation addr) v}};

def set-port (continuation ip fp port) new-port = (continuation ip fp new-port);

def set-fp (continuation ip fp port) new-fp = (continuation ip new-fp port);

def dec-ip (continuation ip fp port) = (continuation (ip⊗1) fp port);
```

Figure 4.5: Simulation of operations performed by the I-structure controller to implement the istr opcode.

Support Opcodes		
	Port	
Support Function	Left	Right
Set Presence bits	set-PB-Empty set-PB-Present set-PB-Delayed set-PB-Deferred set-PB-Error set-PB-Deferred-List set-PB-Spare1 set-PB-Spare2	
Read and Set Presence Bits	Read-set-PB-Empty Read-set-PB-Present ⋮	Write-set-PB-Empty Write-set-PB-Present ⋮
Dispatch on Presence Bits	PB-Dispatch	
Bulk Set Presence Bits (4 word groups)	Bulk-Set-Empty Bulk-Set-Delayed Bulk-Set-Error Bulk-Set-Spare1	Bulk-Set-Present Bulk-Set-Deferred Bulk-Set-Deferred-List Bulk-Set-Spare2

Table 4.2: Support Opcodes

4.8 Emulation on a Monsoon Processor

Since a Monsoon processor's functionality is a super-set of the I-structure controller, a processor can easily emulate it. Each of the I-structure controller opcodes is replaced by a special instruction on the processor. Each instruction is stored in the processor's instruction memory at the address (IP) corresponding to the the opcode number times four. It is multiplied by four because the Pmap is shifted over two places in the IP field; this was done intentionally to leave space between the instructions on the processor. Now, the IP, which on the I-structure controller gives a certain opcode, addresses the instruction on the processor which performs the same operation. This allows transparent processing of requests by either a processor or I-structure controller. This does not say that each will implement them in the same way, but the results will be the same.

The processor does not contain hardware to modify the least significant bits of the IP field which is used by the I-structure controller for generating deferred-return-tag and deferred-value responses. Instead, it performs a sequence of two instructions in which it reads the tag, marks the presence bits as *busy* (a state not used on the I-structure controller),

modifies the tag, and write the tag back in the next instruction. While a location is *busy*, any attempt to access it is forced to try again next time when it will succeed because the location is only *busy* for one cycle. This is a form of busy-waiting, but it is only for one cycle and there is no alternative.

The Monsoon Prototype does contain hardware to flip the least significant bit of the IP, which allowed development of these opcodes. The development was greatly aided by the fact that all of the control was table driven and stored in loadable static memories. It is like micro-code, except that it does not provide any sequencing. All of the opcodes presented in this report have been written and tested on the Monsoon prototype.

4.9 Emulation on the Manchester Dataflow Prototype

The matching functions given for the Manchester prototype in [11] allow I-structures to be emulated, but not deferred lists; busy-waiting is used to defer requests. Each matching function specifies two actions. The first action is taken if a matching token, one with the same tag but opposite port, is found in the matching store. The matching store is where unmatched tokens reside. The second action is taken if no matching token is found. A matching function is specified as {success-action, failure-action}, where success and failure refer to finding a matching token. I-FETCH is implemented by the {Preserve, Defer} matching mode. The Preserve action copies the value found in the matching unit and leaves it in the matching unit (this is the same as finding a *present* value on Monsoon). The Defer action, used when no match is found, sends the token back into the network to try again later, a form of busy-waiting. No deferred lists are generated. The I-STORE instruction uses the {Extract, Wait} matching function. The Wait action stores the incoming value if no match is found. Since the I-FETCH operation never stores a token in the matching section, the I-STORE never finds a matching partner, so the Extract action is never taken.

4.9.1 Locks on Manchester Prototype

Busy-waiting versions of the Non-Busy-Waiting locks from Section 2.3 can be implemented by using {Extract, Defer} for TAKE and {Extract, Wait} for PUT. Since deferring is done by retrying the match later, these locks use busy waiting.

4.9.2 Extensions to Manchester Prototype for Deferred Lists

On the Manchester prototype, a token which matches another token in the matching store with the same tag *and the same port*, is a fatal matching-conflict error. Unfortunately this precludes generating deferred lists, because all of the I-FETCH requests to a location have exactly the same tag and port. What is needed is the ability to control the action performed on a matching conflict, this can be done on Monsoon, but not on the Manchester prototype.

Let us extend the matching functions to include an action for the matching store conflict condition, specified as {success-action, failure-action, conflict-action}. Now I-FETCH uses {Preserve, Wait, Extract} and I-STORE {Extract, Wait, Error}. The first I-FETCH is deferred by being stored in the matching store by the Wait action. On an I-FETCH match conflict, the Extract action sends back the two deferred tags to an instruction which combines them into a deferred list which it sends to the matching store. The Extract in I-STORE is now used to match with a deferred list; the value extracted from the waiting store is a deferred continuation. A deferred I-FETCH in the matching store is equivalent to the Monsoon *deferred* presence bit state. The Error action in I-STORE results in a matching-conflict error, exactly as it did originally. Each of the Monsoon states, *empty*, *present* and *deferred* has a corresponding matching store contents in the Manchester prototype.

There is still one more requirement for implementing deferred lists. The operation performed by each instruction must be based on the matching action. In the case of I-FETCH, the value is returned to the requester for the successful match case, but in the match-conflict case the two deferred continuations must be combined into a deferred list. I-STORE is not a problem because there is no operation performed for either the Wait or Error actions, only on Extract, which must satisfy the deferred request.

The lack of the conflict-matching action and the lack of instruction dispatching prevent the use of deferred lists on the Manchester prototype.

4.10 Presence Maps

Each opcode (Pmap) selects one of sixty-four Finite State Machines (FSM) to control execution for that cycle. The FSMs are called Presence bit maps because they map the current present bit state into a new presence bits state. The states of the FSM are the presence

bit states. One transition is taken on the arrival of each request token. The port of the incoming token along with the current presence bits, pointed to by the token's frame pointer (FP), uniquely determine which transition is taken. The transition specifies a new state for the presence bits, what to write into the value field of the memory, generation of an output token, and statistics collection. The finite state machines can be represented as tables, as they are in Section A.5, as a state transition diagram as in Figure 4.6, or as ID code in Figure 4.5.

In the state transition diagrams, each presence bit state is represented by a circle containing the name of the state. Each transition from one state to the next is drawn as an arrow leaving the initial state and entering the resulting state. The label on an arrow gives the condition of the input variable, port, for which the transition is taken. In Figure 4.6, the transitions labeled "Fetch" are taken when the port is left (port=0) and those labeled "Store" are taken when the port is right (port=1). The state diagrams only show the presence bits changes.

The transition tables are indexed first by Pmap, to choose a FSM, then by the port to choose between read-like and write-like operations. Within each operation, indexing is by the current presence bits. Each entry in the table gives the next value of the presence bits and specifies all of the control signals needed to operate the rest of the I-structure controller. The complete specifications for all opcodes are specified in Tables Table A.2 through A.11 in Section A.5.

4.10.1 I-Structure Presence Maps

The I-FETCH and I-STORE instructions are defined by the presence map represented as a finite state machine state transition diagram in Figure 4.6 and Table A.2. In words, if data is *present* when the I-FETCH arrives then it is returned and the data stays *present*, represented by the arrow, labeled "Fetch," looping back to the *present* state. If the location is *empty* then the I-FETCH is deferred and the state of the location becomes *deferred*. Fetches on *deferred* locations are also deferred and the state stays *deferred*. The *delayed* makes the same transitions as the *empty* state for both I-FETCH and I-STORE; the difference is the generation of the token from the stored delayed trigger.

The transition from the *empty* to *present* state is caused by an I-STORE. A second I-

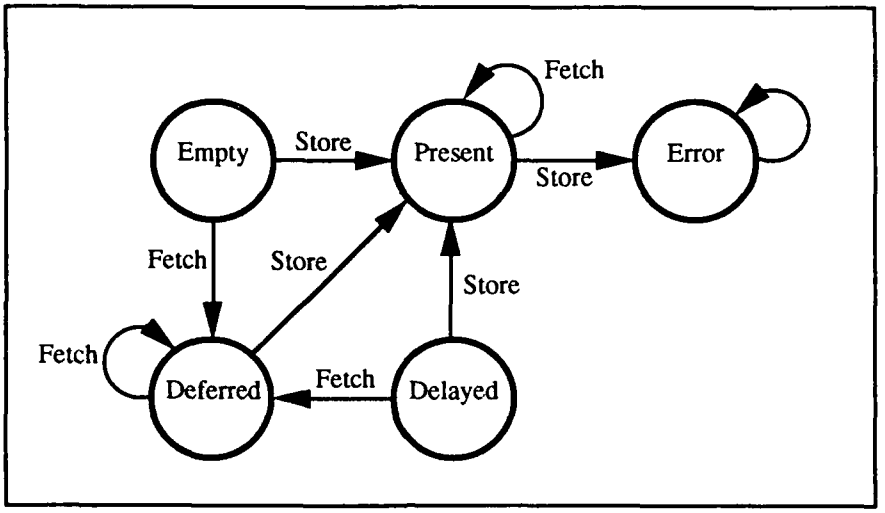


Figure 4.6: State transition diagram for I-FETCH and I-STORE.

STORE on the same location, one already in the *present* state, is a multiple-write error and causes a transition to the *error* state. There are some cases where is it desirable to allow locations to be written more than once, such as in speculation [14]; this is done by changing this one arc to loop back to *present* instead of *error*. Storing into a location on which I-FETCH requests have been deferred (i.e., the location is in the *deferred* state) causes these requests to be satisfied and the location written so that the state becomes *present*. The *error* state, once entered, is not left by either I-FETCH or I-STORE; an imperative operation, such as READ-SET-PB-EMPTY or BULK-EMPTY, is required to clear the error.

The presence bits determine the interpretation of the data stored in the value field of the memory word.. The *present* state signifies a data value. In the *empty* state the location's value is considered to be undefined and the data stored in the memory invalid. When in the *deferred* state the value field contains the return tag of a deferred request. The *delayed* state acts like the *empty* state except that it has a delayed trigger in the value field. The *error* state is used to record errors; the contents of the value field at the time of the error are maintained.

4.10.2 Locks

Locks are accessed by the TAKE and PUT instructions, which are analogous to I-FETCH and I-STORE for I-structures. Figure 4.7 represents the presence bit map for Locks as a state

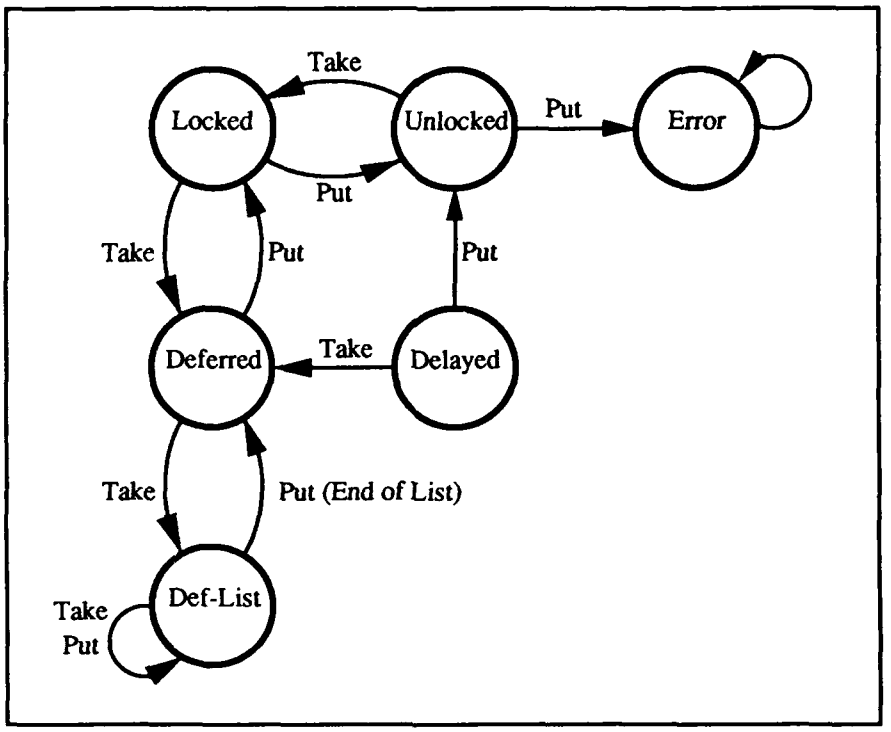


Figure 4.7: State transition diagram for TAKE and PUT.

transition diagram. Table A.4 gives the complete opcode definitions. Since values must be popped off of the deferred lists for Locks, other opcodes may be used by the code handling the deferred lists on the processors.

There are cases when only the value of a lock is needed, not permission to change it. The value of a lock is only valid for its owner, so conceptually a TAKE is used to get the value of the lock and then the same value is immediately PUT back. Using I-FETCH will not work because the deferred lists for locks and I-structures may not be stored the same way. When the lock is *present* in the location (i.e., *unlocked*) the lock the TAKE and PUT can be combined into one operation, with the net result that looks just like an I-FETCH; the value is returned to the requester and the location remains *present*. If the value is not *present*, then the operation acts exactly like a TAKE and is added to the deferred list. The opcode used for this optimized TAKE-PUT sequence is called EXAMINE-LOCK, which is defined in Table A.5.

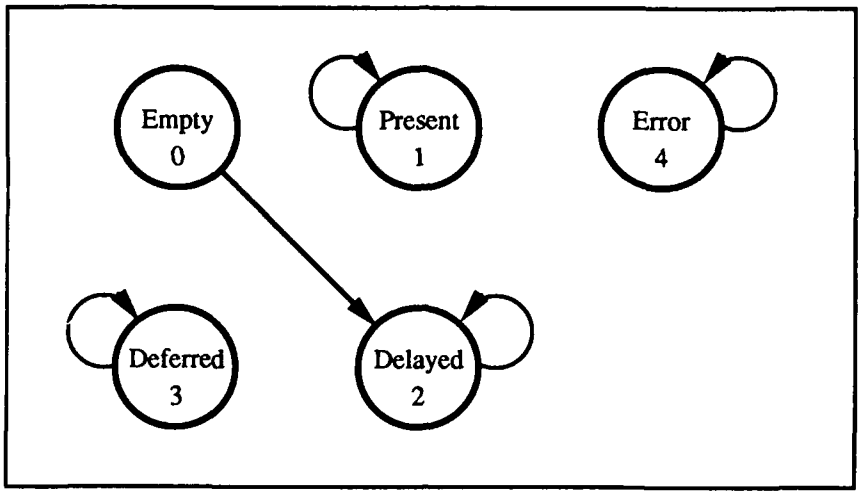


Figure 4.8: Presence map state diagram for STORE-DELAY.

4.10.3 Delayed Triggers

Delayed triggers are extensions to the basic I-structure and Lock operations. A delayed trigger can be stored into either an I-structure or Lock memory location using the STORE-DELAY opcode. The extensions provide for releasing the delayed trigger when the location is accessed by I-FETCH, I-STORE, TAKE or PUT. The state transition diagram for the STORE-DELAY opcode is given in Figure 4.8 and Table A.6. There is no read-like operation for a delayed trigger.

4.10.4 Imperative Operations

Imperative operations ignore the presence bits and only affect the value field of the memory word; the presence bits are not changed. The operation are always performed when they arrive; no synchronization is done. Imperative operations are never deferred, since that is only done as part of synchronization. These operations are intended only for use by the operating system or other low level routines. Figure 4.9 gives the state diagram for the imperative opcodes, which do not change the state of the presence bits; these are READ, WRITE.

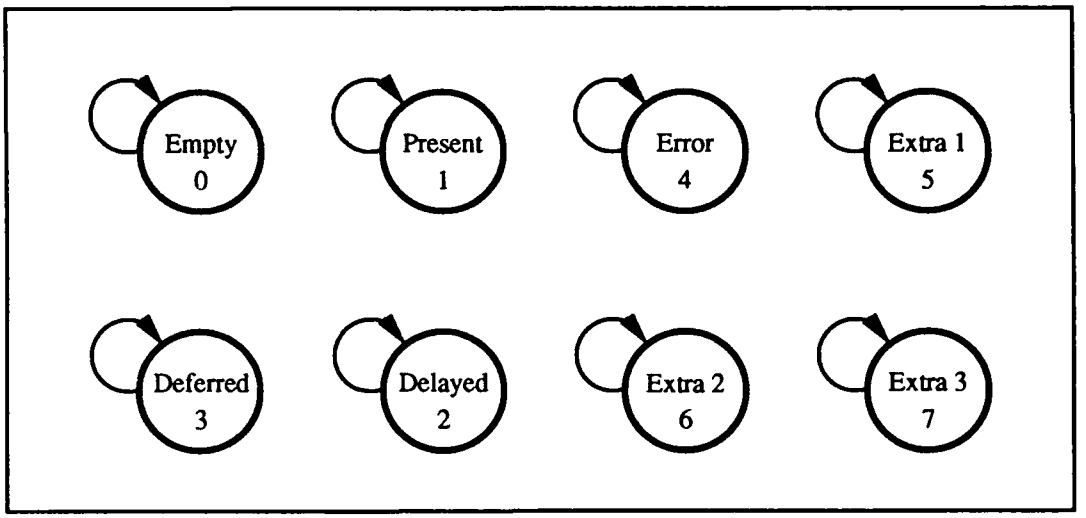


Figure 4.9: State transition diagram for opcodes that do not change the presence bits, such as READ, WRITE and PB-DISPATCH.

4.10.5 Read and Dispatch on Presence Bit State

The PB-DISPATCH instruction reads the location and does a dispatch on the state of the presence bits to return the value read to one of eight possible locations; one location for each of the eight possible presence bit states. To use PB-DISPATCH the continuation sent in the request should have an IP which is a multiple of eight (i.e., the lowest three bits are zero). The I-structure controller replaces the lower three bits of the continuation's IP field with the state of the presence bits of the location requested; this is actually done with an exclusive-ors. The value in the location is sent to this new continuation which results in the value being returned to the instruction with IP = old-IP + *pb*. The value is returned to only one of the eight (one for each possible presence state given presence three bits) possible destination instructions. The presence bits remain the same. See the state diagram in Figure 4.9. The PB-DISPATCH opcode allows the presence bit and value fields of a location to be returned in the same token.

This feature can be used when saving memory images. A PB-DISPATCH is used to read each location's value and presence bits. The lower three bits of the return tag instruction pointer are interpreted as data, the value of the presence bits, instead of as instruction address. The presence bits are saved along with the data returned. A WRITE-SET-PB instruction, which performs the inverse of the PB-DISPATCH, can be used to reload the

memory; it writes the value and sets the presence bits. The state to which the presence bits are set is encoded in the lower three bits of the Pmap of the WRITE-SET-PB opcode family. See Section 4.10.8. Because the Pmap field starts at bit two of the instruction pointer, the the presence bits returned by PB-DISPATCH are not in the same bits of the IP as they are in a WRITE-SET-PB instruction. The presence bits returned from PB-DISPATCH are shifted two bits to the right of those used to set the presence bits in the WRITE-SET-PB instruction.

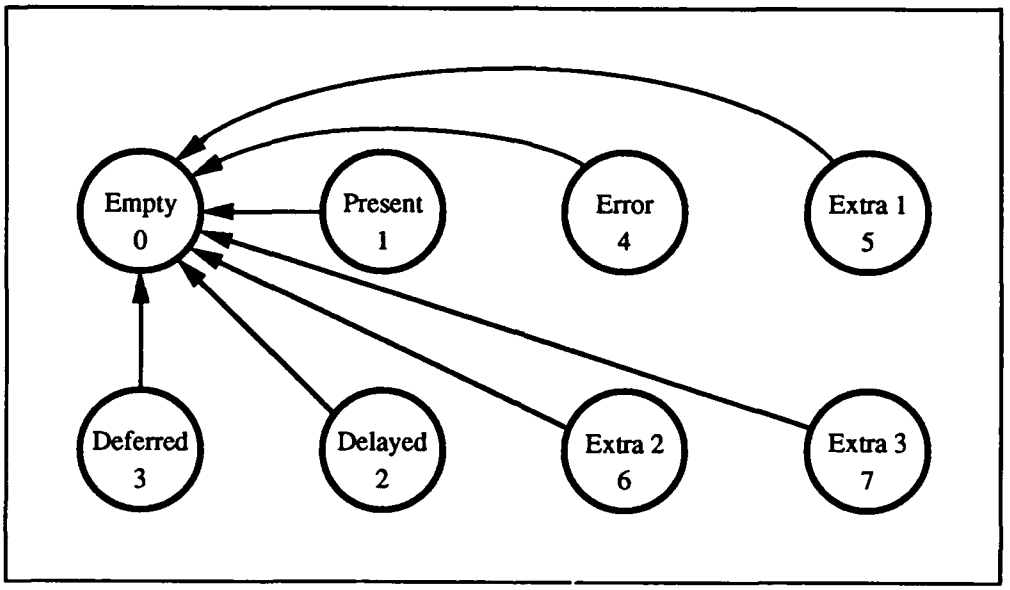


Figure 4.10: Presence map state diagram for SET-PB-EMPTY.

4.10.6 Setting Presence Bits

A class of opcodes is provided to allow the operating system direct control over the presence bits. The Set-PB family of eight opcodes allow the presence bits of a location to be forced to any state, one instruction for each fin ' ·te. The state diagram for the SET-PB-EMPTY instruction is shown in Figure 4.10. These . ·uctions do not perform any synchronization.

4 10.7 Bulk Operations

Bulk operations affect the presence bits of a group memory locations. The purpose is to decrease the time required to initialize memory. There is no restriction on how the words might be grouped. For simplicity of hardware design, the groups are four consecutive words

starting at addresses which are a multiple of four. A bulk operation on any of the words in a group affect all of the words in the group. Because of memory board interleaving, adjacent words on an I-structure board may not be adjacent in the global address space. These instructions are intended only for use by the operation system. The state transition diagrams for bulk operations are the same the SET-PB family of instructions. So BULK-EMPTY has the presence map diagrammed in Figure 4.10. The state transition tables are also the same. The difference is that the upper four opcodes (60, 61, 62 and 63) are special and are always bulk presence bit operations.

4.10.8 Write/Read and Set Presence Bits

The imperative read and write operation, which only affect the value field, and the SET-PB opcodes, which only change the presence bits, can be combined into a new family of opcodes which perform both operations. They are a family of eight opcodes exactly like the SET-PB opcodes and they the state transition diagrams are the same as the SET-PB opcodes, but the affect on data memory and the output tokens generated are those of the READ and WRITE opcodes. The new state transition tables are given in Table A.9. The READ-SET-PB-*state* opcode reads the value field, returns it, and sets the presence bit state to *state*. The WRITE-SET-PB-*state* opcode writes a value into the value field of the memory word and sets the presence bits state to *state*. The opcode numbers are assigned so that the least three bits of the Pmap number are the same as the presence bits to which the state is set. This allows the presence bits to be seen as a sub-field of the Pmap. This is illustrated below in Figure 4.11 below.

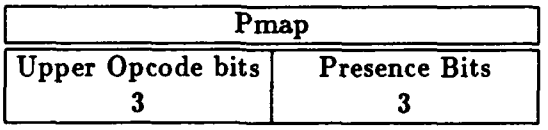


Figure 4.11: Format of presence bits in the SET-PB family of opcodes.

4.11 Handling Errors

There are two different types of errors detected by the I-structure controller: transition errors, such as multiple writes, and hardware errors, such as parity errors.

4.11.1 Transition Errors

Transitions errors are those which are encoded in the Pmap FSMs as errors. They are reported by generating a token with a special error tag. The error tag is stored in a register on the controller and is used only as a source for the output tokens tag. The frame pointer part of the error tag is the frame pointer of the input token, the address of the currently accessed word. The data portion of the error token is selected by the FSM. The error continuation is expected to generate an exception on a processor. Which processor is determined by the PE field of the error tag stored in the error register at system initialization. The error register can only be written using the scan path from the VME bus. By using the exclusive-oring functions available on the output tag, eight different error instruction pointers can be generated.

Which state transitions cause what types of errors is completely programmable in the Pmap FSMs. On the diagrams above, any transition not explicitly defined is an illegal transition, including transitions from undefined states. Writing to a present I-Structure location is another type of error. Each type of error is distinguished by exclusive-oring a different value into the low order bit of the error tag's instruction pointer.

4.11.2 Hardware Detected Errors

Errors detected by the hardware, such as parity errors and out of range addresses, are handled by the VME interface. The action take when a hardware error condition is raised is determined by the VME interface. It can halt the I-structure controller, cause a VME interrupt, or both.

Chapter 5

Future Work and Improvements

We have presented three methods of synchronizing access to memory: I-Structures, Non-Busy-Waiting Locks, and Delayed Triggers. A non-synchronizing access method, referred to as Imperative, was also included. These methods provide a simple programming model of the memory system that does not require the programmer to think about timing and synchronization. Hardware support for synchronization reduces the associated overhead. Two hardware primitives are needed: presence bits and deferred lists. By using distributed deferred lists, the extra hardware needed for synchronization is minimized. Much of the hardware on the design presented is required because the interconnection is a network and not a bus. If the memory did not do any synchronization, the three presence bits, much of the decode memory, and the form token multiplexor (which is wide) could be removed, but that is less than twenty percent of the hardware.

The reason for the design is to provide a large memory for the Monsoon dataflow computer, which must support I-structures; that has been done. As a side benefit, the same synchronization primitives can be used to support the other, less frequently used, synchronization methods. The argument for needing hardware is only made of I-structures, because that is the fundamental method of accessing memory in the Monsoon dataflow computer and the basis for all data structures in the ID language.

The synchronization and deferred list storage methods presented in Chapters 2 and 3 are not limited to dataflow computers. Any processor can use I-structures if it uses split-phase memory transactions, can tolerate long latency, store a continuation in a single word, and accept returned data in any order. For synchronization to be efficient, the processor should

be able to efficiently switch contexts to hide latency. Under these constraints, any method of execution can be used on the processors.

A hardware implementation designed for the specification included in Appendix A is being built by the Micro Computer Division of Motorola in Tempe, Arizona. It will be the memory board for the Monsoon dataflow system. The goal of a practical design shaped the thinking in this report. The throughput of the I-structure controller was matched to the token delivery rate of the PaRC network used in the Monsoon system. Future performance enhancements are likely to be needed. This chapter discusses ideas on ways to increase throughput, reduce latency, and reduce network traffic. Other areas that may need improvement or more study are also noted.

Other dataflow research projects are using memory systems that employ I-structure synchronization. The Epsilon-2 [4] project at Sandia National Laboratory in Albuquerque, New Mexico includes plans for an I-structure controller that stores the deferred lists locally. Research at IBM on the EMPIRE dataflow processor project also uses an I-structure controller controller with local deferred lists. The work is based on Bob Iannucci's PhD thesis [8].

5.1 Size of Deferred List Space?

A more detailed study of the pattern of deferred reads in real applications is needed. This study is complicated by the fact that the execution order of a program can change if the order or time in which reads are satisfied is changed. A simple program trace, like those used in conventional cache simulations, does not provide sufficient information because the order of execution is affected by the latency and ordering of memory operations. The only way to get accurate simulation results is to run programs on a full system simulator, including both processors and I-structure controllers. The kind of information needed to evaluate design decisions is the distribution of deferred requests as a function of time, for each I-structure controller. Also histograms of the deferred list lengths would help determine the amount of time required to satisfy deferred lists.

Since the first deferred read can be stored in the memory word, no extra memory is required. Only further deferred reads consume deferred space. For local deferred list space,

the fragmentation of deferred list space across multiple controllers needs to be taken into account. If one controller overflows its deferred list space, the whole system will be affected, even if all the other controllers have free deferred list space.

5.2 Areas for Improvement

The throughput rate of the current design is matched to the rate of the PaRC network. As the network bandwidth increases, or a direct connection is made to the processor, as described below, the required throughput will increase. One way to achieve this is with faster memories. Another is to access multiple memory banks in one cycle. Using the single chip I-structure controller as described in Section 5.7 is a way to implement multiple bank accesses.

Splitting the presence bit access and data memory accesses into two pipeline stages could help throughput, at the cost of increased latency.

5.2.1 Reducing Latency

Latency was sacrificed for simplicity of design in some cases. Most noticeably, idle periods are a full cycle, which on average adds half a cycle of latency to every request. This is due to the fact that PaRC can start delivering a token on any 20 ns clock edge. If that is during an idle cycle the cycle must end before a cycle to process the new token can start. This only occurs when a token arrives at an idle I-structure controller. When there are tokens waiting in the input queue, they are processed without intervening idle cycles, but each is delayed by the amount of time the first token waited to be processed, because the second token can not be started until the first finishes. Shorter idle cycles would decrease this startup time.

The whole token is shifted out of the network before a memory cycle is started. It is possible to start earlier. Since the network delivers the token at a known rate, a memory cycle can be started as soon as the address, the frame pointer, has been shifted out. The instruction pointer and data type are needed next to determine the presence map and type map. The type of the input value is needed next to determine the Type Code for use in the presence map decoding. The first need for the token's value is to generate parity. If the type map were eliminated, only the first half of the token, the tag, would be needed to do

everything up to writing data memory. In the current Monsoon token format the order of fields is not significant, but is properly ordered for this purpose; the address is the first field of the token to arrive. The type map is not currently used, but was provided to maintain compatibility with the Monsoon processor and for future experimentation. Overlapping the generation of the output token and shifting it into the network has the problem that the data read from memory can be used as the output tag, which is the first thing shifted into the network.

5.3 Direct Connection Between I-Structure Controller and Processor

Another method of connecting I-structure controllers and processors is to pair them and provide a direct connection between them. This could reduce the latency and increase the communication bandwidth. The peak input rate to the I-structure controller would then be the maximum rate at which the processor could generate tokens, 10 Million token/sec on the current processor. The expected sustained rate would be lower, but could be higher than the network. This would require higher throughput from the I-structure controller.

A problem can arise with this method of connection. Since every I-structure controller shares a PaRC chip with a processor, it is possible for blockage of the network to an I-structure controller to block a processor. Once this happens the machine can dead-lock. The processors are designed to always act as sinks for tokens so that they never block the network. The processor can use its token queues to augment its input queues. The I-structure controller does not have a large input queue, so it stops accepting tokens from the network when its output to the network becomes blocked. If a cycle in the network is generated in which the output of an I-structure controller is blocked because of a blocked input of an I-structure controller, then a cycle has been formed and the system could become dead-locked. This problem is avoided in the current system by disallowing I-structure controller and processors from sharing PaRC chips. A method for the processor to queue requests for the I-structure controller to which it is paired could solve this problem.

5.4 Cacheing Deferred Lists on I-Structure Controller

The problem with storing deferred lists locally on each I-structure controller is handling the overflow of local deferred list space. What should be done when the local free list is exhausted and another deferred read arrives? Pre-allocating deferred list space in activation frames solves the allocation problem, at the cost of extra network traffic. These two methods can be combined by cacheing deferred requests locally when possible, but falling back on the frame allocated space when the cache is full. This requires the same amount of storage to be allocated in the frames, but will reduce network traffic and latency.

This also brings back the problem of variable length service times to satisfy the deferred read requests stored in the cache. More than one token is generated for some input tokens, so input will be queued during this time, increasing their latency. To help this, the cache can limit the length of deferred lists it stores.

The complexity of the I-structure controller and hardware is increased at a savings of network traffic and overhead instructions executed on the processor. This should also reduce the latency of satisfying a deferred read.

5.5 Interleaved Memory Banks

Currently the only type of interleaving used is to interleave multiple I-structure boards. This is simply done by spreading a structure across multiple I-structure boards. Interleaving of banks can be done locally on an I-structure controller to increase throughput by overlapping the memory recovery time of one bank with an access to a different bank. This requires separate address and data paths to memory and independent RAS (row address strobe) and CAS (column address strobe) signal generation. Bank conflict hardware is also needed to detect two consecutive accesses to the same bank. A single chip I-structure controller for each memory bank could be a simple way to implement this idea.

5.6 Cacheing Fetches On Processors

A further method of reducing network traffic is to cache network traffic locally on each processor. The cache coherence problem only occurs when memory is deallocated. This

is due to the same fact that make I-structures invariant to ordering, locations are never redefined. Timing is not an issue because I-structures are already immune to timing. A simple write through cache will be consistent for all I-structures. Special consideration needs to be given to Lock locations.

The continuation of a Fetch to an uncached location is cached locally and a request to return data to the cache is sent over the network. Future Fetches of that location, before the value is returned, are deferred in the local cache by storing their continuations. When a value is returned to the cache it is sent to all of the deferred Fetches, whose tags are removed from the cache. The value is stored in the cache. Fetches to a cached location are satisfied from the cache. The frame allocated method of storing deferred lists can be used to store multiple deferred lists in the cache.

Stores are written through the cache; the store is always sent to the network. If the location had cached deferred Fetches then these are satisfied. The memory system also returns the value to the cache because of the deferred read stored there, this is ignored by the cache. An alternative is to just send all stores to the network directly, bypassing the cache, and let the caches deferred fetch fill the cache. This is simpler, but has a higher latency; the value must travel to the I-structure controller and then back to the processor. That is the same latency that would be seen without the cache.

There are two types of data to flush from the cache, values and deferred fetches. Values can simply be thrown away, they are also stored in memory. Deferred fetches need to be written back to memory. A deferred fetch is flushed by reforming the original token and issuing it to the network. Flushing deferred fetches might not be worth the complication.

The cached data must be keyed by the address of the location fetched, not the continuation of the fetch.

5.7 Single Chip Controller

Based upon the pin out requirement, it looks possible to put all of the I-structure controller, except the dynamic memory, in a single chip. Four basic interfaces are needed: PaRC, presence bit memory, data memory, and control. See Table 5.1. A direct connection to the PaRC has 16 bit datapaths in each direction, plus handshaking. All of the timing and

control functions for the dynamic memories can be put inside the I-structure controller chip. Since the address lines of dynamic memories are multiplexed, the address lines from the I-structure controller chip can be multiplexed in the same way, halving the number of address pins. The RAS, CAS and WE (write enable) signals are the only control signals needed for the memories. Separate address and control lines are used for the presence bit and data memories to allow different timing. Separate input and output paths to the presence bit memories allow faster read-modify-write cycles at a small cost because the widths are small, four bits. Every access to the presence bits are read-modify-write cycles. The large data width, 73 bit, requires a bi-directional bus to the data memory. Other control signals, such as clock, error reporting, halting and scan path are not require a large number of pins.

Section	Signal	I/O	Pins	Total
PaRC	Data In	I	16	
	Data Out	O	16	
	Wait	I	1	
	Ready	O	1	
	Dout CLK	O	1	
	CSACK	I	1	36
Presence Bits	Address	O	11	
	CAS, RAS, WE	O	3	
	PBCURR	I	4	
	PBNEXT	O	4	22
Data Memory	Address	O	11	
	CAS, RAS, WE	O	3	
	Data	I/O	73	87
Control	50 MHz Clock	I	1	
	CE, SE	I	2	
	SDI	I	1	
	SDO	O	1	5
Grand Total				**150**

Table 5.1: Signal Pins required for a Single Chip I-structure Controller

The second consideration is the complexity of the internal circuits. Here building a minimal hardware design is again beneficial. Most of the complexity is due to wide data paths. The table look up method of control decoding is unlikely to be practical in a single chip design. A fixed instruction set would allow this to be encoded directly in hardware.

Using the single chip controller in Figure 5.1, an I-structure controller can be built with:

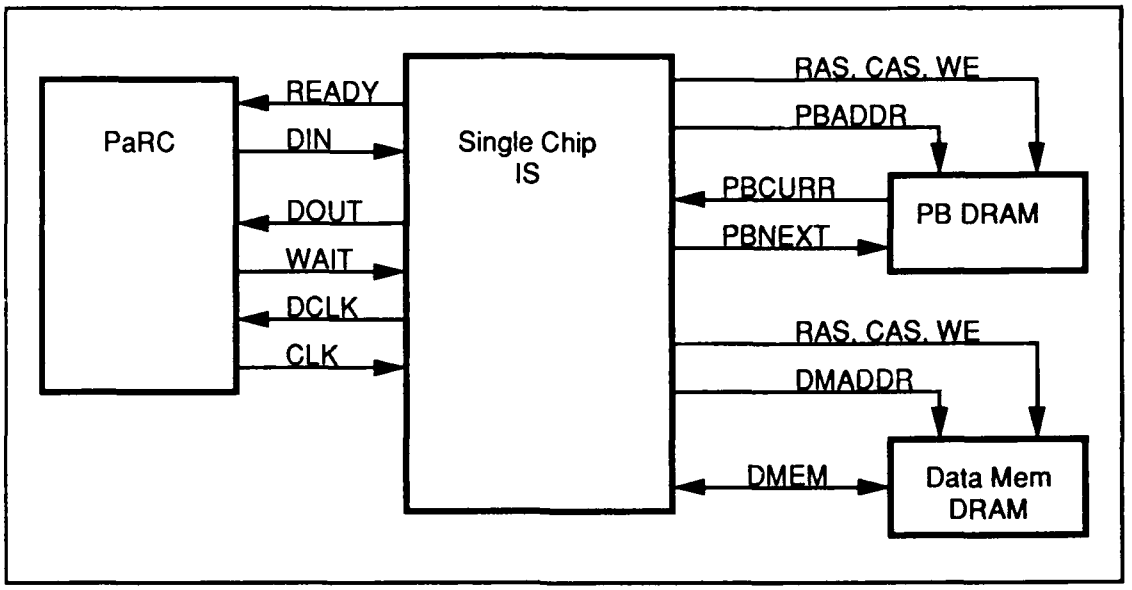


Figure 5.1: Block Diagram of Controller using I-structure controller chip.

1 Data link chip, 1 PaRC chip, 1 I-structure controller chip, 77 DRAMs and some address buffers. Each I-structure controller chip manages one bank of memory. Having a separate controller for each bank allows all banks to be access simultaneously.

Appendix A

Implementation Specification for a Minimal I-Structure Controller

This is a specification of the minimal functionality needed to implement I-structures and TAKE/PUT locks. It will be as close to a detailed implementation as possible. The I-structure controller (IS) is designed to be used in a system with Monsoon Processing Elements (PEs). A working knowledge of I-structures and the Monsoon processor is assumed in this appendix.

The IS uses dynamic memory to store data. It acts as a memory board in a Monsoon system. Each address has a value (72 bits), Presence Bits (3 bits), and parity. It is a separate node on the PaRC network. Each IS has up to 16M words of address space. All deferred list are allocated in PE Frame store.

A.1 Design Limitations

The limitations specified for the design are listed below. These limits were imposed to simplify the design of the IS while still preserving its functionality.

1. Process 1 input Token per cycle
2. No Structure-to-Structure communication
3. For Each input Token processed:
   (a) One presence state transition
   (b) One Read, Write or Exchange to data memory

(c) Output 0 or 1 Tokens

4. All state must be scannable

Problems encountered while designing previous versions of I-structure controllers are described below. These are the reasons for many of the design limitations listed above.

Over Flow of Deferred List Storage Space When deferred lists are stored locally on each IS then the space for deferred lists must be pre-allocated locally on the IS. What to do when the space is exhausted is a difficult problem. Solution: Pre-allocate space for the deferral of each read in the frame that requested the read; referred to as frame allocated deferred lists.

Input FIFO Over Flow Since deferred lists are of potentially unbounded length, the time to service a locally stored deferred list when the value is written is unbounded. This leads to the problem of the input FIFO overflowing and blocking the network. Allowing only one output token per input token eliminates this problem.

Network Dead-Lock When two IS boards communicate directly and have bounded input FIFOs, there is the potential for the network to become dead-locked. Therefore, this type of communication is disallowed.

A.2 Functional Requirements

The two basic operations supported are I-structures (Fetch and Store) and Locks (TAKE and PUT). Other support operations, such as reading presence bits, are implemented, but only as absolutely necessary.

The following operations are supported by the IS: (37 Presence maps)

- I-FETCH and I-STORE
- TAKE and PUT
- SET-PB (8)
- Bulk (Group of 4 words) (8)

- Read&Clear a Statistics counter (8)
- READ and WRITE (imperative)
- PB-DISPATCH (imperative)
- WRITE-SET-PB (imperative) (8)
- STORE-DELAY

A.3 Hardware

A.3.1 Token Format

The token format is that defined by Monsoon:

Token			
Tag-part		Value-part	
TYPE	TAG	TYPE	VALUE
8	64	8	64

Figure A.1: Token Format

The IP field is used as the instruction, not the address of an instruction as it would be on a PE. The MAP and PE fields are not used and are passed unchanged in all cases. The format of the fields in a tag are:

Tag				
-	MAP	IP	PE	FP
1	7	24	8	24

Figure A.2: Format of Token Tag-Part Fields

The IP field of the tag-part of the incoming token is used to encode the function to be performed by the IS. The PMAP field is the opcode for the IS. The PORT bit is also used to decode the operation to be performed.

IP			
PORT	-	PMAP	-
1	15	6	2

The PORT bit has exactly the same meaning as it does on the Monsoon processor when each IS opcode is viewed as an instruction. For I-structure operations the port bit distinguishes between fetch and store operations. Right port for write operations and left port for read.

Port Encoding	
PORT	Port
0	Left
1	Right

A.3.2 Interface to PaRC

The input interface with PaRC is exactly the same as Monsoon's, a serial to parallel shift register and a 512 deep FIFO. The output section is different in that it does not have an output FIFO. The parallel to serial shift register is loaded directly from the output registers.

The control of the interface between PaRC and the input FIFO is all local. The only information needed is the fullness of the input FIFO. This control is done with the clock output from PaRC. The input section and output section of the input FIFO are in different clock domains which will require resynchronization of the EMPTY signal from the FIFO.

Flow control of the output section is achieved by making the input FIFO look empty when PaRC asserts WAIT. At that time the token being shifted is completed, as well as the on-going operation in the other sections of the pipeline. If a new output token is generated during the cycle in which WAIT is asserted, then no token is taken from the input FIFO, even if one is present. VME tokens may still be processed since they do not generate output tokens. Note, this may not be true of all VME accesses. If the operation that cycle does not generate an output token then new tokens maybe processed until one does produce an output token. If PaRC is still asserting WAIT at that time then no more tokens may be taken from the input FIFO.

A.3.3 PaRC Network

The PaRC chip is a 4x4 packet routing chip designed for the Monsoon computer system. It is implemented in a single LSI gate array. It is designed to handle a fixed size packet equal to one Monsoon token. Each of the four input and output ports is 16 bits wide. The chip

operates at 50 MHz, which is one 16 bit word every 20 ns. Each packet comprises a 16 bit header, 72 bit tag, 72 bits of data and a 32 bit CRC for a total of 192 bits or twelve 16 bit words. So a new packet can arrive every 240 ns (4.17 MHz). Packets can be transmitted at the same rate, independently of packet arrivals. Packets can arrive and be transmitted on any 20 ns clock edge.

A.3.4 Memory format

The format of the data stored at each memory address is:

MEMORY WORD				
PB PARITY	PBCURR	DM PARITY	TYPE	VALUE
1	3	1	8	64

A.3.5 Presence Bit Memory

There are 3 presence bits per memory word plus 1 bit of presence bit parity. Presence bits are stored in 4 banks of dynamic memory. The multiple banks allow bulk operations on four consecutive presence bits starting at an address which is a multiple of four.

A presence state transition consists of reading the current presence bit state, looking up the next state and writing the new state back into memory. The read-write cycle of the DRAM chips is used to provide valid read data from the access time until CAS is deasserted, even during the write. The presence bits are latched to keep the decode output signals valid until the end of the cycle.

The presence bits read from memory (PBCURR), the Presence Map number (PMAP) and the Type Code (TC) form the decode memory address. This address is used to look up the next value for the presence bits (PBNEXT), including a parity bit. The decode address is also used to address other decode memories to control the Data Memory and Form Token sections. These are viewed as separate decode memories, even though they could be combined, because the timing requirements on them may be different. The outputs of the decode memories are valid as long as the presence memory outputs are valid; the other address bits are registered. The presence bits are latched until the end of the cycle.

A.3.6 Data Memory

Each data word is 72 bits wide, 64 bits of value and 8 bits of type. Extra bits are added to each word for parity checking. Every memory cycle is a read-write cycle, the same as the presence bits. A read is always done; the write is conditional, controlled by a bit in decode memory. This allows an exchange to be performed in one cycle and simplifies generation of the RAS and CAS timing signals.

The lower three bits of the IP and the PORT of the value written to memory can each be individually inverted using the signals DMXOR and DMPOP respectively. It is implemented with XOR gates. The signal DMADJ is the parity of the 4 bits in the signals DMXOR and DMPOP. It is XORed with the parity of V_{in} to adjust for the effect of the XORs mentioned above. This allows most of the parity to the computed before these signals are valid.

Data Mem Port Op Encoding	
DMPOP	Data Memory Port Written
0	V_{in} port
1	$\overline{V_{in}port}$

A.3.7 Multiple Memory Banks

Data memory has four banks of memory interleaved the same way as the presence bit memory. These banks are used only to expand the memory, not to overlap accesses. There is the option of using either 1Mbit or 4 Mbit DRAMS for a total of either 4 M- or 16 M-words of storage. That is either 36 Mbytes or 144 Mbytes. Other configurations of banks, such as 1 or 2, could have been made an option to provide a wider selection of memory sizes. The choice to only allow four banks was made to simplify the design.

The Data memory uses address bits 0 and 1 for bank selection. This results in interleaving on a word by word basis. The eleventh address line to the DRAM is multiplexed between address bits 20 and 21; it is only used by the 4 Mbit chips.

Data memory banks are selected by controlling both RAS and CAS to each bank separately. The PB banks use only CAS for bank selection, this results in a refresh of the same row in the other banks. Some power is wasted, but the number of control signals is reduced. For both PB and Data Memory, the same WE is used for all four banks. A separate WE is used for PB and Data memory.

A.3.8 Refresh

512 refresh cycles are required every 8 ms for 1 Mbit DRAMs (1024 every 16 ms for 4 Mbit) which is once every 15.625 *μsec*, which is 65.1 240 ns cycles. This is simplified to 64 240 ns cycles, every 15.36 *μsec*. To allow for variations in clock speed, a refresh is done more frequently than every 64 cycles. A 6 bit counter is used to generate the signal to request a refresh cycle.

CAS before RAS refresh is used to simplify hardware. The CAS before RAS timing test is not used. A system level refresh test is depended upon to catch non-functional refresh counters in the memory chips. The refresh RAS signal is staggered from the normal RAS signal to reduce noise.

To avoid interfering with token processing, refreshes are hidden on banks which are not being accessed. Each 15.6 *μsec* refresh period is divided into four sub-periods, one sub-period for each bank. At the beginning of each sub-period a request to refresh the bank is made. If during a cycle the bank is accessed by a token, then the refresh is postponed. Otherwise, the bank is refreshed and the refresh request is removed. If a refresh request is still pending during the last cycle of a sub-period, then it becomes a refresh demand; a refresh is forced by not processing any tokens. Using this method the only event that will cause a refresh to steal a cycle is consecutive accesses to the bank attempting to refresh during the whole sub-period. This does introduce some variance into the time between refreshing the same location. The refresh could occur at the beginning of the sub-period the first time and the end of the sub-period the next. So all 512 (or 1024) refreshes must be done in 8 ms - 4 *μsec* (one sub-period).

A.3.9 Parity

Parity is used to detect memory errors. No correction is done. One parity bit is used for the 3 presence bits, as in the PE, and one parity bit is used for the 72 bits of data.

Only one parity bit is needed for the 72 bit data word, since the probability of two errors in one word is very low. More bits could be used if the time to generate or check the parity is too long. Generating the parity is not a problem since the write data is available (at least most of it) 100 ns before it is written. Error detection time is not critical since a parity

error causes a fatal error, but this error must be detected before the end of the cycle, while the data needed to track down the offending error is still in scannable registers.

The hardware to generate and detect a parity error is an XOR tree with 72 inputs for generation and 73 inputs for detection. The detection tree has an extra input for the parity bit.

A Parity error will halt the IS and generate a VME interrupt

A.3.10 Decrement Exchange Cycles

To implement frame allocated deferred lists a Decrement-Exchange operation is required. When a location that is deferred is fetched, the Decrement-Exchange is used to extend the deferred list. This operation modifies the incoming value, which should be a tag, and sends the value read from memory to the modified tag and stores the modified tag in memory. The decrement on the prototype PE was implemented with a single XOR of the least significant bit of the IP field. This is generalized to be an XOR of the lower three bits of the IP field with three control bits from decode memory. This modifies the value written into data memory. The XOR is always done. If no change is desired then the three control bits are set to zero.

A.3.11 Bulk Presence Bit Operations

To deallocate and reallocate memory, it must be cleared (presence bits set to the empty state). To do this efficiently requires the ability to clear more than one word per input token. The Bulk-PB operation sets the presence bits of four consecutive words, starting at an address which is a multiple of four, to the same value. All four presence bit banks are written simultaneously, using early write cycles so that none of the memories output data. The PBCURR bus is pulled up to a stable state, to give the decode memory a stable address. Using pull ups, PBCURR value will be all ones. The PBNEXT output of the decode memory is written to all four banks.

Bulk operations are assigned to a fixed set of presence maps and decoded directly from the Pmap number. For compatibility with the PE, Pmaps 60 to 63 are defined as bulk

operations. With PORT, this gives 8 possible bulk operations. Only Bulk-Clear is used on the prototype.

A.3.12 Decode Memory

Decode memory is static memory which contains all of the controls for the other sections of the IS. It is a simple table look-up addressed by the presence bits, Presence Map number, Type Code, and port. See section A.3.5 and section A.3.13. No sequencing can be done.

The bits used for addressing are tabulated below. The fields of the decode memory are tabulated in the next three sections.

Decode Memory Address			
PMAP	TC	PBCURR	PORT
6	2	3	1

Data Memory Decode

Memory Decode Fields				
ENWE	DMPOP	DMXOR	DMADJ	PBNEXT
1	1	3	1	4

ENWE Enables write to Data Memory

DMPOP Controls modifying the port of the value written to memory.

DMXOR The value XORed into the IP field of the value written into Data Memory.

DMADJ Parity adjustment for Data Memory parity. Parity of DMXOR and DMPOP.

PBNEXT The next state of the presence bits. Written to PB mem. Includes parity.

Form Token Decode

Form Token Decode Fields					
TSEL	VSEL	TPOP	TXOR	ENTOK	CS
2	2	2	3	1	1

TSEL Selects the output tag. See table in section A.3.14.

VSEL Selects the output value. See table in section A.3.14.

TPOP Controls the formation of the port in the output tag

TXOR The value XORed into the IP field of the output tag

ENTOK Enables issuing formed token to the network.

CS Request Circuit Switch acknowledge for generated token

Statistics Counter Decode

Statistics Decode Fields			
STAT	SOP	CSEL	FCZ
3	2	1	1

STAT Selects statistics counter from set of eight.

SOP Operation to perform on statistics counter selected.

CSEL Select Color to select statistics counter set.

FCZ Force Color to be zero.

A.3.13 Type Map

The same style of type map used on Monsoon is used on the IS. The only difference is that the Type Code (TC) used on the IS is only two bits instead of the four used on Monsoon. Each instruction has its own type map which maps the 8 bits of input value type to a 2 bit Type Code (TC). The type code can be viewed as another input on the presence map state diagrams. The type map is implemented with a single SRAM.

How the type map will be used has yet to be determined. There is no known need for it. It is included for generality and possible future use.

A.3.14 Form Token

The Form Token section composes the output token from the input tag, value, the value read from memory, an exception tag register and the statistics counters. The inputs to Form Token can not be combined in pieces, each is taken as a whole unit. The ENTOK field determines if the token will be output to the network.

Table for Form Token Tag and Value selection multiplexors:

Output Tag Selection	
TSEL	Output Tag
0	V_{in}
1	Mem_{out}
2	ERR_{tag}
3	Reserved

Output Value Selection	
VSEL	Output Value
0	V_{in}
1	Mem_{out}
2	T_{in}
3	Statistics

After the output tag is chosen, 3 bits (TXOR) are XORed into the lower 3 bits of the IP field, to allow the same decrement function that can be performed on data written to memory, as described in section A.3.10.

Further functionality is provided to modify the PORT bit of the output tag. The four options are to pass the port unchanged, invert it, force it to Left or force it to Right. The changes to the tag port are determined by the TPOP decode field.

Output Tag Port Encoding	
TPOP	Output Tag Port
0	T_{out} port
1	$\overline{T_{out}port}$
2	Left
3	Right

The ENTOK field determines if the token composed is issued to the network.

Token Output Enable Encoding	
ENTOK	Action
0	No token output
1	One token issued to network

A.3.15 Exception Tag Register

The Exception Tag register is a 58 bit register that holds a tag, except for the FP. The value of the register and the FP of T_{in} together form ERR_{tag}, which is used as the output tag for all transition errors. By using the TXOR field this can be modified to generate 8 distinct tags. The register can only be accessed by the scan path. It is used as one of the sources for the output tag in the Form Token section.

This allows both the address and the value of an error to be sent to an error handler running on a processor. This feature allows double write errors to be checked for equality before reporting an error. Double writes that do cause an error can report both the value and address involved.

ERR_{tag}						
Exception Register						Input Tag
TYPE	-	MAP	PORT	IP	PE	FP
8	1	7	1	23	8	24

A.3.16 Statistics Counters

The same statistics counters are used as on Monsoon. The STAT field selects a counter from a set of counters selected by the COLOR field of one of the input tags. The CSEL field chooses the color from either the input tag type or the input value type. The FCZ can be used to force the color to zero. The operation performed on the selected counter is determined by the SOP field. The counter selected can be read by selecting the Statistics output with VSEL.

Color Selection Encoding	
CSEL	Type Selected for Color
0	Color taken from type of input Tag
1	Color taken from type of input Value

Statistics Operation Encoding	
SOP	Statistic Function
0	Pass
1	Increment
2	Decrement
3	Clear

Type Field Interpretation		
user definable	COLOR	PRIV
4	3	1

Only COLOR is used.

It would be desirable to keep the maximum value a counter reached. This is not done.

A.3.17 VME Interface

A simple VME interface is required to access the scan paths and initialize the PaRC chip. The interface should be a stripped down version of the PE VME interface. Direct access to IS memory over VME is provided by inserting tokens into the input registers and reading values from the memory output bus. This interface can be removed if board space is limited.

The VME interface provides:

- Scan path reading and writing
- PaRC Initialization
- Configuration register access
- Single stepping control
- IS memory access (Optional)

Status signals readable from VME:

- Input FIFO Full
- Input FIFO Empty
- Output FIFO Full (PaRC Wait)
- PB Parity Error
- Data Memory Parity Error
- Statistics Counter Overflow
- Halted
- Error Token generated (Set if TSEL = ERR_{tag})

Control signals generated by VME:

- Halt (Force input FIFO to appear empty to IS)
- Single Step
- Disable Token output
- Back Load (For writing decode memories)
- Write Enable signals for Decode memory
- Write Enable signals for Type Map memory
- Write Enable signals for Route Generation memory
- Enable VME interrupt on Parity error
- Read Input FIFO (into input register)

Single stepping the IS board will cause the token in the input register to be processed. If a refresh cycle is requested at the same time, it will take priority. If the input FIFO is not empty at the end of the single step cycle then a new token will be loaded into the input register. The output token will remain in the output register after the single step cycle, even if the ENTOK signal would have caused it not to be issued. The output token is also sent to the network.

Note: The IS does not need to know its PE number; that is determined by how the network routes tokens.

VME Direct Memory Interface (Optional)

The IS memory is memory mapped into the VME address space in the same manner that the PE frame store is mapped. 16 Mwords of IS memory requires 128 Mbytes of VME address space. Writes are done by loading a type, value and Pmap into the input token register. The FP is decoded from the VME address and the input token register is the source of the next token instead of the input FIFO. For reads, only the Pmap and FP are loaded into the input token register and the token is inserted. The output of the memory is loaded into the VME output register, along with PBCURR. The VME output register holds the 64 bits of value, 8 bits of type and 3 bits of presence bits. Three 32 bit accesses are needed for VME to read all of these values, two for the value and one for the type and presence bits. The VME output register is loaded for every VME token inserted.

This interface allows any token to be inserted into the IS, as if it came over the network. By using the I-STORE Pmap, writes can be done to deferred read lists and will be handled properly. Inserting a VME token can cause a token to be issued to the network.

A.3.18 Scan Path

The same style of scan path is used in the IS as Monsoon. The scan path is used to read and write all controller state, including loading the decode memories. The scan path is accessible only by VME. The scan path is clocked at 4.17 MHz. The registers on the scan path are listed in Table A.1.

Scan Path Registers

Input Token Register	144 bits
Output Token Register	152 bits
Exception Tag Register	58 bits
Observer on PBCURR	4 bits
Observer on Type Code (TC)	4 bits
Back loading on Decode Memory	28 bits

Table A.1: Scannable Registers.

The statistic counters are not directly scannable. Observers are needed to control TC and PBCURR and all of the decode output signals, in order to load the memories. For route generation, an observer is needed on the outputs for back loading. The address to route generation comes from the output register.

Scan loaded decode memories:

- Type Map
- Decode Memory
- Route Generation

A.3.19 Estimate of Parts Count

- 308 DRAMs (4 banks)
- Approximately 300 other chips.

A.4 Timing

The IS operates on a 240 ns cycle time, to match the PaRC token delivery rate. All of the processing for a token is done in one cycle. This includes the presence bit read-modify-write cycle, a read, write or exchange cycle on data memory and forming an output token. The formation of the output token can be overlapped with the recharge time of the dynamic ram.

Since PaRC can deliver a token on any 20 ns boundary, arriving tokens must be synchronized to the IS 240 ns clock. If no token is available at the start of a cycle, then it is an Idle cycle, even if one arrives during the cycle. On average this will increase the latency by half the Idle cycle time for a token that arrives at an idle IS. This does not decrease the throughput of the IS.

A.4.1 Idle Cycles

Idle cycles result from an empty input FIFO. During an Idle cycle the RAS and CAS signals are inhibited, along with the generation of an output token. Also reporting of parity errors is inhibited because the data input to the parity checkers is not valid. An Idle cycle is the same length as all other cycles and is not affected by the arrival of tokens during the cycle.

A.4.2 Memory Timing

The PB and Data Memory will have independent timing. The critical path is from FP, through the PB mem to PBCURR, through Decode Mem to the DM control signals which set up the data to be written to Data memory. Separate timing allows the PB memory, which is smaller (16 chips), to return data before the data memory. The Data memory is given more time for address propagation and delays writing until almost the end of the cycle. How late the write can be done is constrained by the need to read data before the end of the cycle so that parity can be checked and an output token formed.

A.4.3 Latching PB Memory Output Data

The data from the PB memory (PBCURR) is latched to simplify timing. The data is latched sometime after it is read and held valid until the end of the cycle. Thus, all control signals

decoded from the presence bits are valid until the end of the cycle.

The Data Memory outputs are valid until the end of the cycle because CAS is held active until the end of the cycle.

A.4.4 Scanning and Refresh

Refresh cycles and scanning will not interfere with each other. Scanning only affects scannable registers, which can only change the address presented to the DRAM array. A refresh affects only the DRAMs and their timing controls; address lines are ignored during refresh. Scanning is only done while the IS is halted, which means it is running Idle cycles. Idle cycles don't cause RAS and CAS cycles.

A.4.5 Clock Domains

The clock for the IS is generated from the 50 MHz clock for PaRC, to keep them synchronous.

Since the propagation delay through the FIFO is longer than 20 ns, the status signals from the FIFO to the IS (INEMPTY) are resynchronized to the 50 MHz clock. Each is passed through two 40 ns registers to avoid metastability problems.

The Ready and Wait signals from PaRC are also asynchronous and are synchronized in the same manner.

A.4.6 Timing Generation

The timing controls for the memories and FIFOs are generated by gating free-running clock signals with 240 ns periods.

Care must be taken in generating the equations for the gating to prevent glitches.

A.5 Software

Software support on the PE and microcode on the IS are needed for the IS to function. Functionality was moved into code running on the PE to simplify the IS hardware.

A.5.1 I-Fetch/I-Store Opcode

I-FETCH and I-STORE operate on I-structure locations. One IS opcode is used to implement both I-FETCH and I-STORE. Left port signifies an I-FETCH and right port signifies an I-STORE. Below is the state transition table for the presence bit operations:

I-Fetch State Transition Table			
State	Next State	Write	Token Issued
Present:	Present		$< V_{in} >< Mem_{out} >$
Empty:	Deferred	V_{in}	
Deferred:	Deferred	$V_{in} - 1$	$< V_{in} - 1 >< Mem_{out} >$
Delayed:	Deferred	V_{in}	$< Mem_{out} >< T_{in} >$
Error:	Error		$< ERR_{tag} >< V_{in} >$

I-Store State Transition Table			
State	Next State	Write	Token Issued
Present:	Error		$< ERR_{tag} >< V_{in} >$
Empty:	Present	V_{in}	
Deferred:	Present	V_{in}	$< Mem_{out} >< V_{in} >$
Delayed:	Present	V_{in}	$< Mem_{out} >< T_{in} >$
Error:	Error		$< ERR_{tag} >< V_{in} >$

Table A.2: **I-Fetch** and **I-Store** State Transition Table

A.5.2 Imperative Read/Write Opcode

The Imperative Read and Write operations ignore the presence bit state and do not change it, only the data value is affected. As with the other operations, one opcode is used for both. Left port signifies a Read and right port signifies a Write.

Read State Transition Table			
State	Next State	Write	Token Issued
$\forall x$	x		$< V_{in} >< Mem_{out} >$

Write State Transition Table			
State	Next State	Write	Token Issued
$\forall x$	x	V_{in}	

Table A.3: **Read** and **Write** State Transition Tables

A.5.3 Take/Put Opcode

TAKE and PUT are used to access Lock locations. TAKE requests the lock and PUT releases the lock. One IS opcode is used to implement both TAKE and PUT. Left port signifies a TAKE and right port signifies a PUT. Below is the state transition table for the presence bit operations:

Take State Transition Table			
State	Next State	Write	Token Issued
Present:	Empty		$< V_{in} >_L < Mem_{out} >$
Empty:	Deferred	V_{in}	
Deferred:	Deferred	V_{in}	$< V_{in} >_R < Mem_{out} >$
Delayed:	Deferred	V_{in}	$< Mem_{out} >< T_{in} >$
Error:	Error		$< ERR_{tag} >< V_{in} >$

Put State Transition Table			
State	Next State	Write	Token Issued
Present:	Error		$< ERR_{tag} >< V_{in} >$
Empty:	Present	V_{in}	
Deferred:	Empty		$< Mem_{out} >_L < V_{in} >$
Delayed:	Present	V_{in}	$< Mem_{out} >< T_{in} >$
Error:	Error		$< ERR_{tag} >< V_{in} >$

Table A.4: **Take** and **Put** State Transition Tables

A.5.4 Examine-Lock Opcode

The EXAMINE-LOCK opcode is used to examine the value of a lock. It is used when the value of a lock is needed, but not further ownership; the value obtained can be used, but

not modified. EXAMINE-LOCK acts like a TAKE followed immediately by a PUT. This is optimized to the point that when the lock is *present* the operations acts like a read (it Takes and PUTs back the lock in one operation). Other than the *present* state, it is exactly like the TAKE operation. The opcode's function is defined by Table A.5.

Examine-Lock State Transition Table			
State	Next State	Write	Token Issued
Present:	Present		$< V_{in} + 1 >_L < Mem_{out} >$
Empty:	Deferred	V_{in}	
Deferred:	Deferred	V_{in}	$< V_{in} >_R < Mem_{out} >$
Delayed:	Deferred	V_{in}	$< Mem_{out} >< T_{in} >$
Error:	Error		$< ERR_{tag} >< V_{in} >$

Table A.5: **Examine-Lock** State Transition Table

A.5.5 Store-Delay Opcode

The Store-Delay opcode writes a delayed trigger into an I-structure or Lock location. A separate opcode is used in place of Write-Set-PB-Delayed to handle the cases where the I-FETCH or TAKE arrives before the Store-Delay; the trigger is then released immediately. The possible state transitions are listed in Table A.6

Store-Delay Transition Table			
State	Next State	Write	Token Issued
Present:	Present		$< V_{in} >< T_{in} >$
Empty:	Delayed	V_{in}	
Deferred:	Deferred		$< V_{in} >< T_{in} >$
Delayed:	Error		$< ERR_{tag} >< V_{in} >$
Error:	Error		$< ERR_{tag} >< V_{in} >$

Table A.6: **Store-Delay** State Transition Table.

A.5.6 Set Presence Bit Opcodes (8)

The Set Presence Bit opcodes are actually a family of eight opcodes, one for each possible presence state. Table A.7 gives the state transition table for the opcodes; *state* represents any of the eight possible presence bits state. For example, SET-PB-EMPTY sets the presence

bits state to empty, independent of the previous state. Each opcode sets the presence bits to one state without affecting the data value. The port bit is ignored. No tokens are output.

Set-PB-*state* State Transition Table			
State	Next State	Write	Token Issued
$\forall x$	*state*		

Table A.7: Set Presence Bits State Transition Table.

Set-PB-Empty State Transition Table			
State	Next State	Write	Token Issued
(0)	Empty		
(1)	Empty		
(2)	Empty		
(3)	Empty		
(4)	Empty		
(5)	Empty		
(6)	Empty		
(7)	Empty		

Table A.8: **Set-PB-Empty** State Transition Table.

A.5.7 Bulk Presence Bit Opcodes (8)

The Bulk Presence Bit opcodes are exactly like the Set Presence Bit opcodes, except that the presence bits of four words are affected.

A.5.8 Read/Write-Set-PB Opcodes (8)

The READ-SET-PB-EMPTY and WRITE-SET-PB operations are actually a family of eight opcodes, similar to the SET-PB operations. These operations are a combination of the Read/Write opcodes and the SET-PB opcodes. Table A.9 gives the state transitions for these opcodes; *state* can be any one of the eight possible presence bits states, giving a family of eight different opcodes. On a WRITE-SET-PB operation both the data value and the presence bits are written. On a READ-SET-PB-EMPTY operation, the presence bits are set and the value of the location is read and returned. The WRITE-SET-PB opcode is designed

for loading data into memory; it is the counter part to the PB-DISPATCH opcode. It could be used by a disk controller to load a memory image.

Read-Set-PB-*state* Transition Table			
State	Next State	Write	Token Issued
$\forall x$	*state*		$< V_{in} >< Mem_{out} >$

Write-Set-PB-*state* Transition Table			
State	Next State	Write	Token Issued
$\forall x$	*state*	V_{in}	

Table A.9: **Read-Set-PB** and **Write-Set-PB** State Transition Tables.

Read-Set-PB-Empty Transition Table			
State	Next State	Write	Token Issued
(0)	Empty		$< V_{in} >< Mem_{out} >$
(1)	Empty		$< V_{in} >< Mem_{out} >$
(2)	Empty		$< V_{in} >< Mem_{out} >$
(3)	Empty		$< V_{in} >< Mem_{out} >$
(4)	Empty		$< V_{in} >< Mem_{out} >$
(5)	Empty		$< V_{in} >< Mem_{out} >$
(6)	Empty		$< V_{in} >< Mem_{out} >$
(7)	Empty		$< V_{in} >< Mem_{out} >$

Write-Set-PB-Empty Transition Table			
State	Next State	Write	Token Issued
(0)	Empty	V_{in}	
(1)	Empty	V_{in}	
(2)	Empty	V_{in}	
(3)	Empty	V_{in}	
(4)	Empty	V_{in}	
(5)	Empty	V_{in}	
(6)	Empty	V_{in}	
(7)	Empty	V_{in}	

Table A.10: **Read-Set-PB-Empty** and **Write-Set-PB-Empty** State Transition Tables.

A.5.9 PB-Dispatch Opcode

Reads the data value and sends it to one of eight return addresses based on the presence bit state. The presence bits are XORed into the return tag's IP field. The presence bit

state os not changed. This can be used to read the data and presence bits in one operation, possibly by a disk controller. The presence states are represented as decimal numbers in parentheses because the meaning of the state is unimportant.

PB-Dispatch State Transition Table			
State	Next State	Write	Token Issued
(0)	(0)		$< V_{in} \oplus 0 >< Mem_{out} >$
(1)	(1)		$< V_{in} \oplus 1 >< Mem_{out} >$
(2)	(2)		$< V_{in} \oplus 2 >< Mem_{out} >$
(3)	(3)		$< V_{in} \oplus 3 >< Mem_{out} >$
(4)	(4)		$< V_{in} \oplus 4 >< Mem_{out} >$
(5)	(5)		$< V_{in} \oplus 5 >< Mem_{out} >$
(6)	(6)		$< V_{in} \oplus 6 >< Mem_{out} >$
(7)	(7)		$< V_{in} \oplus 7 >< Mem_{out} >$

Table A.11: **PB-Dispatch** State Transition Table.

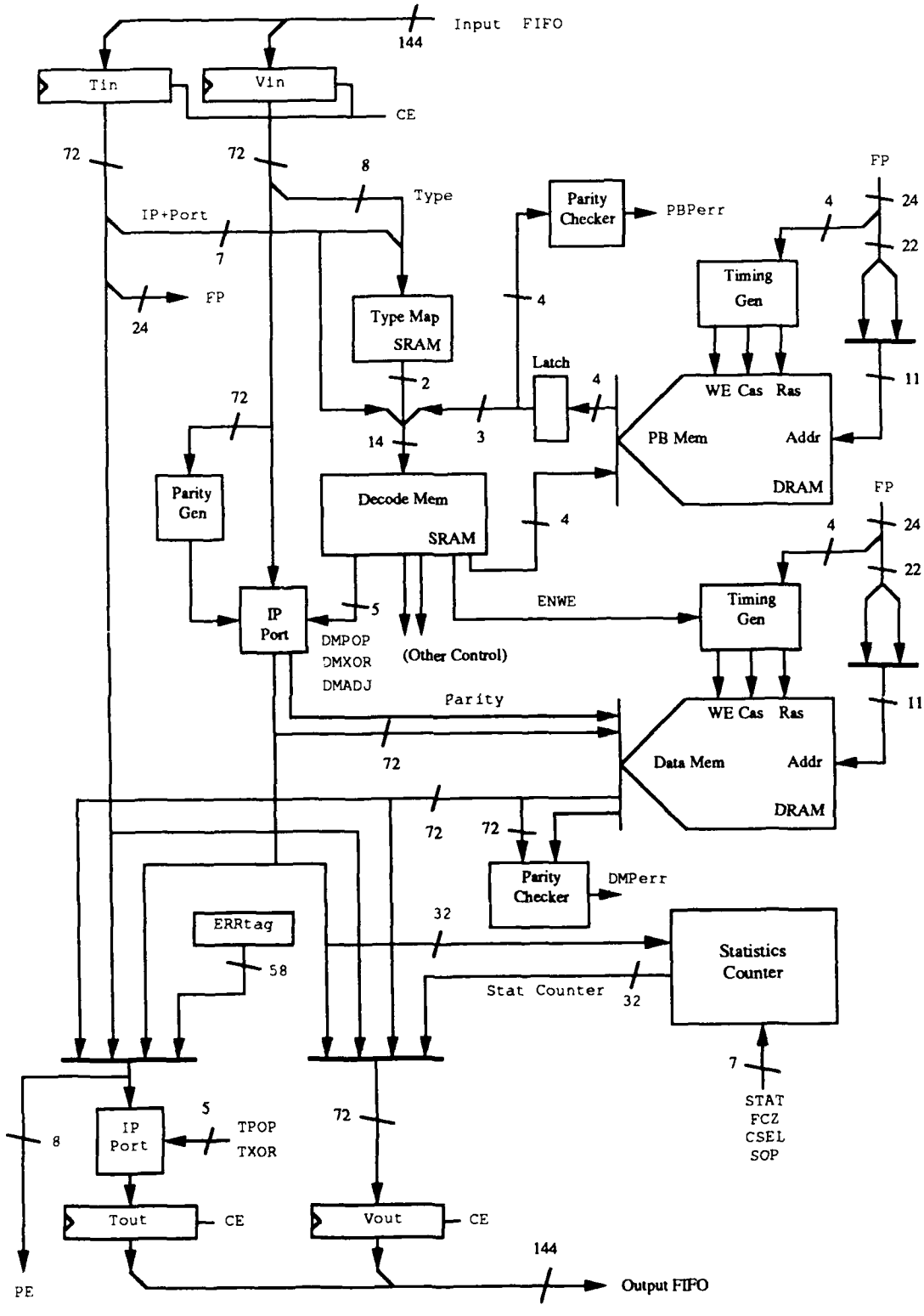


Figure A.3: Data Path for I-structure Controller

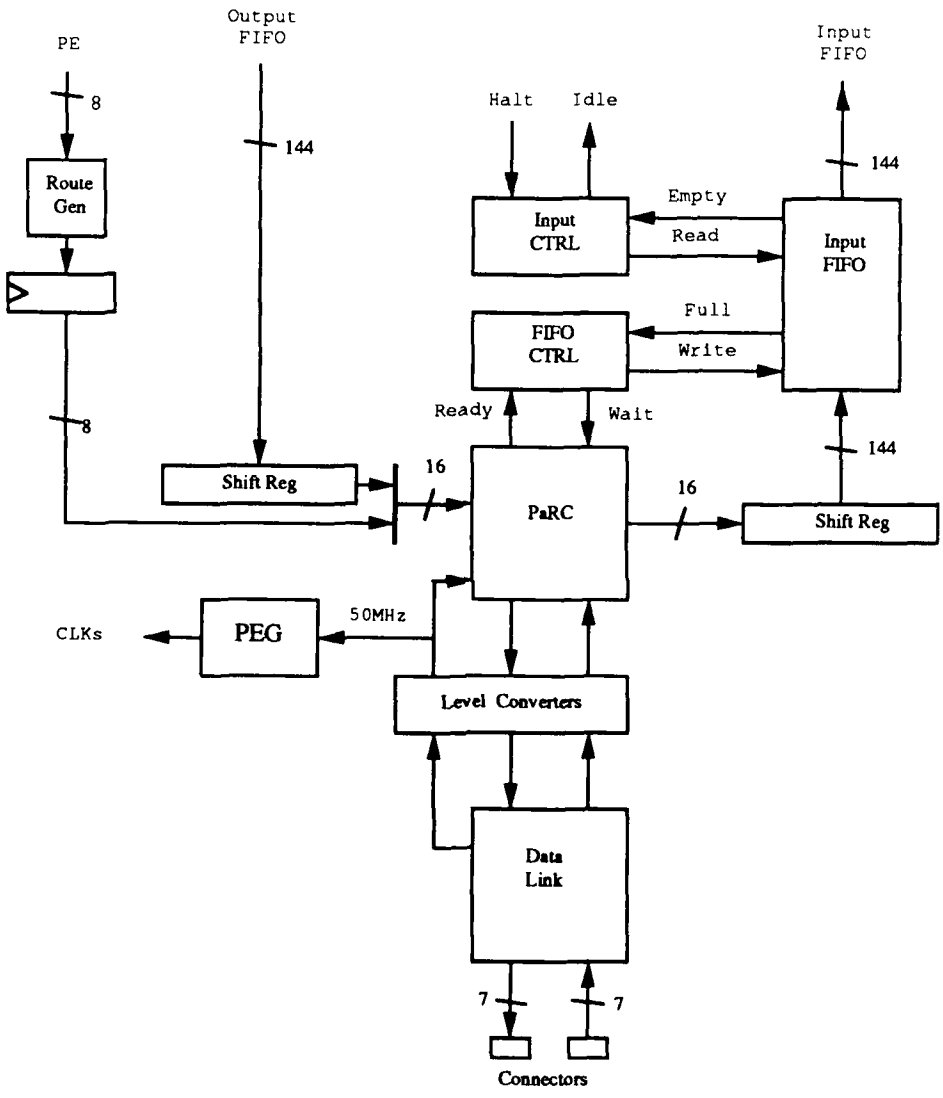


Figure A.4: Input/Output Section of Minimal I-structure Controller

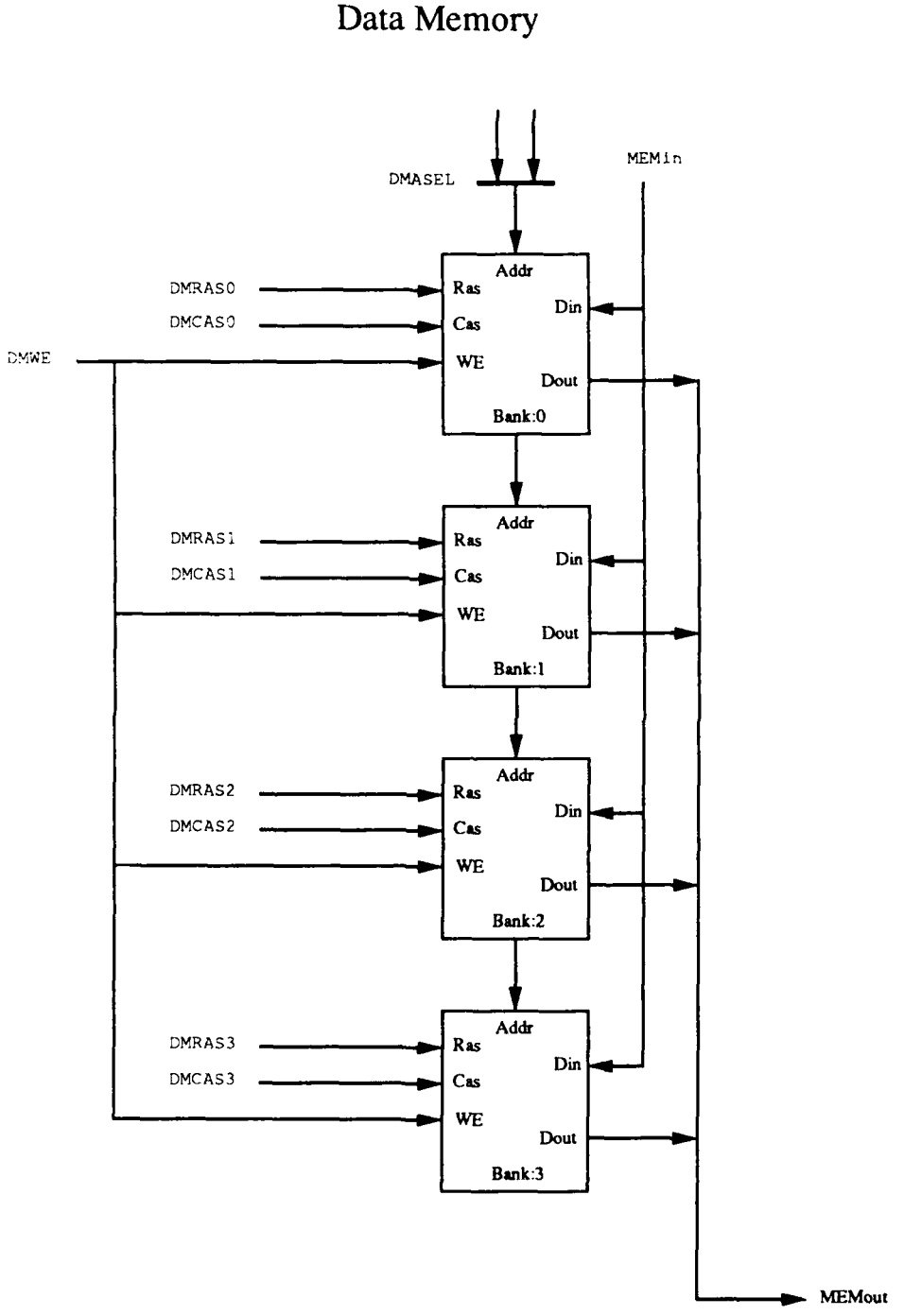


Figure A.5: Data Memory Banks of Minimal I-structure Controller

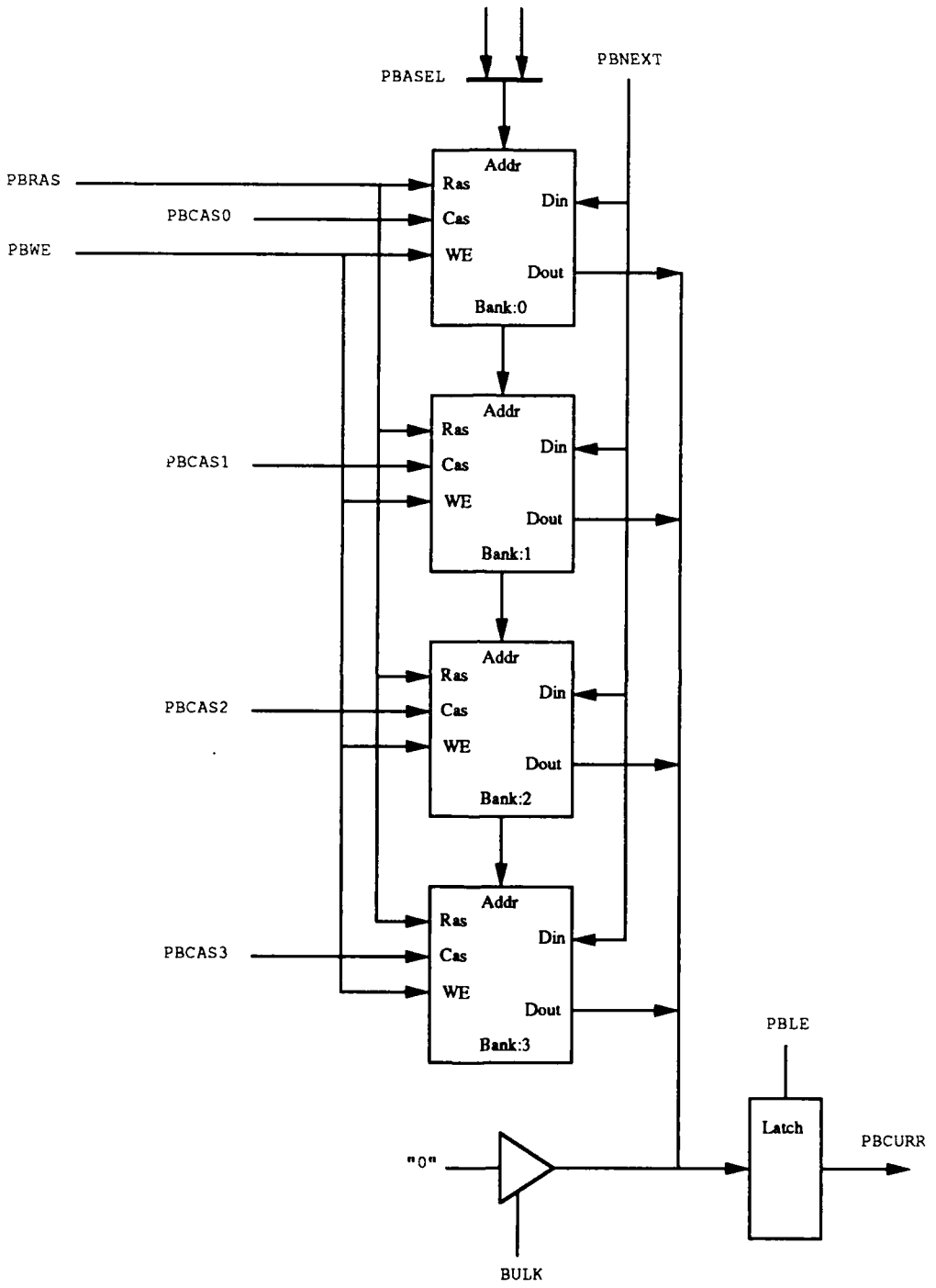


Figure A.6: Presence Bit Memory Banks of Minimal I-structure Controller

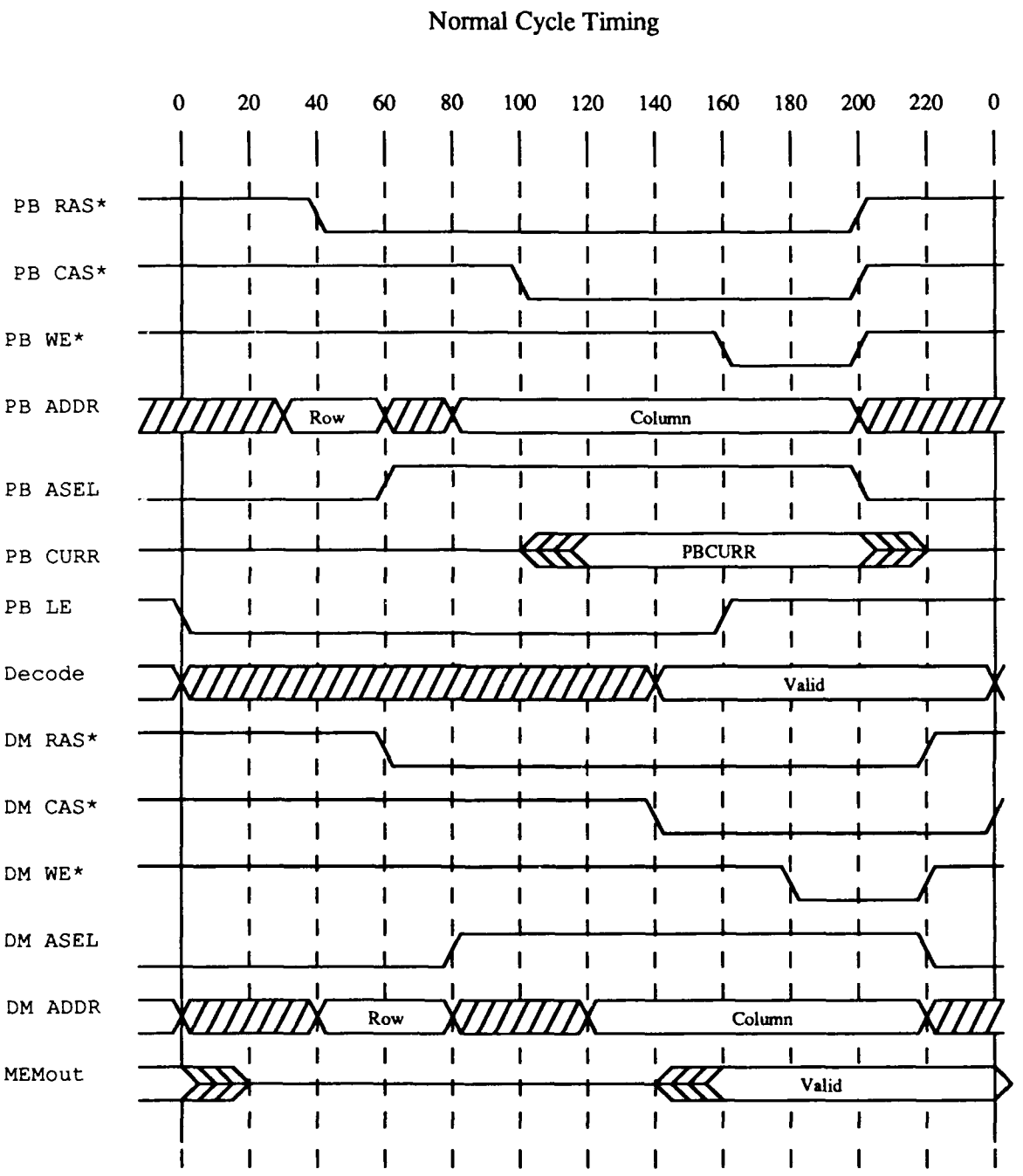


Figure A.7: Timing Diagram of a Normal Memory Cycle

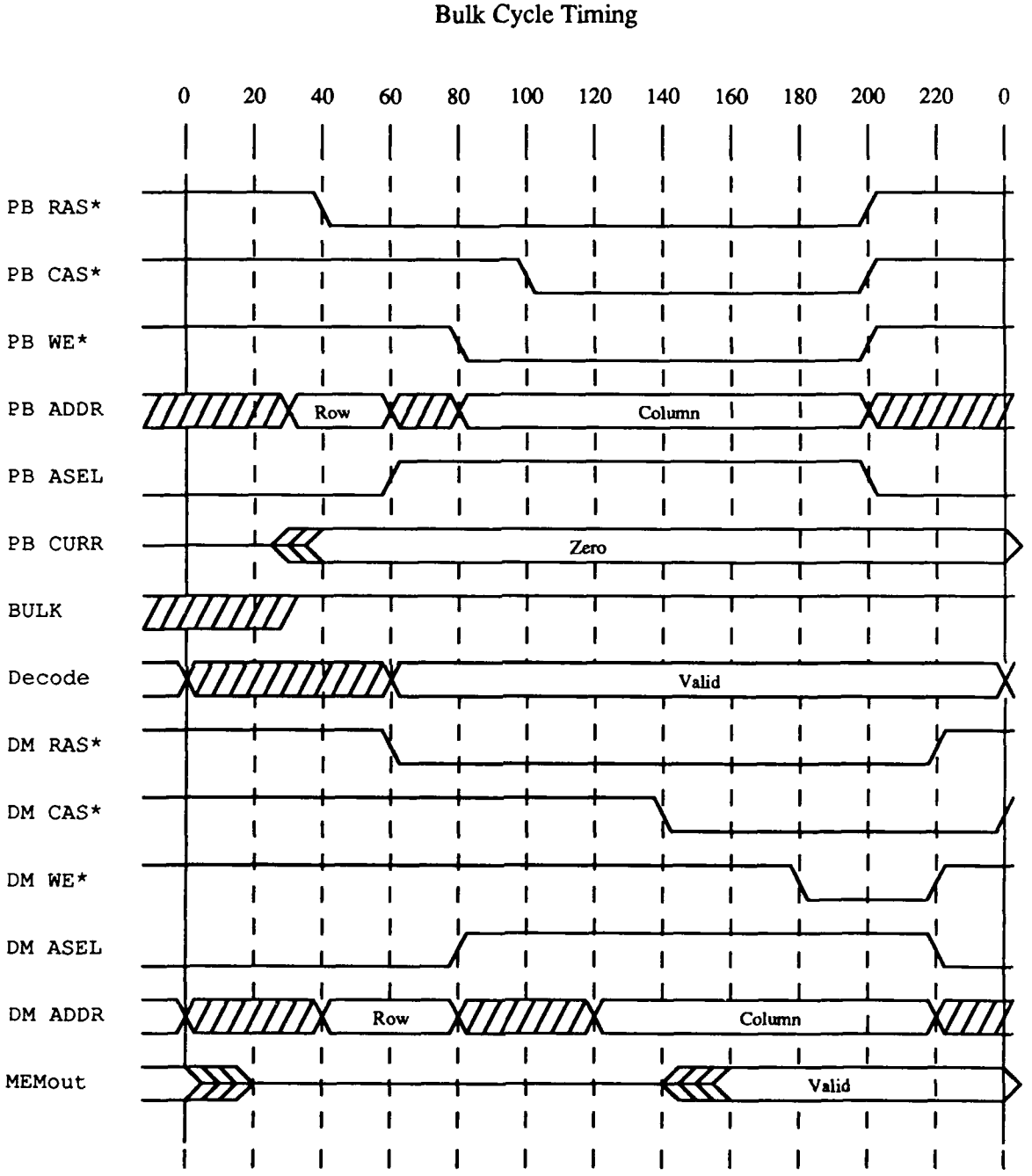


Figure A.8: Timing Diagram of a Bulk Presence Bit Operation Cycle

Bibliography

[1] Arvind and David E. Culler. Dataflow architectures. In *Annual Review of Computer Science, Volume 1*, pages 225–253. Annual Reviews, Inc., Palo Alto, California, 1986.

[2] Arvind and Robert E. Thomas. I-structures: An efficient data type for functional langauges. Technical Report LCS/TM-178, MIT, September 1980.

[3] Paul Barth and Righiyur S. Nikhil. Supporting state-sensitive computation in a dataflow system. Computation Structures Group Memo 294, M.I.T. Laboratory for Computer Science, March 1989.

[4] V.G. Grafe and J.E. Hock. The epsilon-2 hybrid dataflow architecture. In *COMPCON-90*. Sandia National Laboratories, 1990.

[5] Steven K. Heller. An I-structure memory controller (ISMC). Master's thesis, Massachusetts Institute of Technology, May 1984.

[6] Steven K. Heller. *Efficient Lazy Data-Structures on a Dataflow Machine.* PhD thesis, MIT, February 1989.

[7] K. Hiraki, S. Sekiguchi, and T. Shimada. System Architecture of a Dataflow Supercomputer. Technical report, Computer Systems Division, Electrotechnical Laboratory, 1-1-4 Umezono, Sakura-mura, Niihari-gun, Ibaraki, 305, Japan, 1987.

[8] Robert Alan Iannucci. A dataflow / von neumann hybrid architecture. Technical Report LCS/TR–418, MIT, May 1988.

[9] Christopher F. Joerg. Design of a packet switched routing chip for the dataflow supercomputer. Master's thesis, Massachusetts Institute of Technology, May 1987.

[10] Robert M. Keller, Gary Lindstrom, and Suhas Patil. An architecture for a loosely-coupled parallel processor. Technical Report UUCS-78-105, Univeristy of Utah, October 1978.

[11] C.C. Kirkham. *The Manchester Prototype Dataflow Computer Basic Programming Manual*, 3rd edition, April 1982.

[12] R. S. Nikhil and Arvind. Id Nouveau. Computation Structures Group Memo 265, M.I.T. Laboratory for Computer Science, 1986.

[13] Gregory Micheal Papadopoulos. *Implementation of a General Purpose Dataflow Multiprocessor.* PhD thesis, MIT, August 1988.

[14] Richard Mark Soley. *On the Efficient Exploitation of Speculation Under Dataflow Paradigms of Control.* PhD thesis, MIT, May 1989.

OFFICIAL DISTRIBUTION LIST

DIRECTOR — 2 Copies
Information Processing Techniques Office
Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, VA 22209

OFFICE OF NAVAL RESEARCH — 2 Copies
800 North Quincy Street
Arlington, VA 22217
Attn: Dr. Gary Koop, Code 433

DIRECTOR, CODE 2627 — 6 Copies
Naval Research Laboratory
Washington, DC 20375

DEFENSE TECHNICAL INFORMATION CENTER — 12 Copies
Cameron Station
Alexandria, VA 22314

NATIONAL SCIENCE FOUNDATION — 2 Copies
Office of Computing Activities
1800 G. Street, N.W.
Washington, DC 20550
Attn: Program Director

HEAD, CODE 38 — 1 Copy
Research Department
Naval Weapons Center
China Lake, CA 93555